RICE UNIVERSITY

A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism

by

Yi Guo

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar

Vivek Sarkar, Chair
E.D. Butcher Professor of Computer Science

Keith D. Cooper

Keith D. Cooper
L. John and Ann H. Doerr Professor of Computer Science

William N. Scherer III,
Faculty Fellow
Dept. of Computer Science

Lin Zhong
Assistant Professor
Dept. of Electrical & Computer Engineering

Rice University, Houston, Texas

Aug, 2010

A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism

Yi Guo

Abstract

Recent trend has made it clear that the processor makers are committed to the multi-core chip designs. The number of cores per chip is increasing, while there is little or no increase in the clock speed. This parallelism trend poses a significant and urgent challenge on computer software because programs have to be written or transformed into a multi-threaded form to take full advantage of future hardware advances.

Task parallelism has been identified as one of the prerequisites for software productivity. In task parallelism, programmers focus on decomposing the problem into sub-computations that can run in parallel and leave the compiler and runtime to handle the scheduling details. This separation of concerns between task decomposition and scheduling provides productivity to the programmer but poses challenges to the runtime scheduler.

Our thesis is that work-stealing schedulers with adaptive scheduling policies and locality-awareness can provide a scalable and robust runtime foundation for multi-core task parallelism. We evaluate our thesis using the new Scalable Locality-aware Adaptive Work-stealing (SLAW) runtime scheduler developed for the Habanero-Java programming language, a task-parallel variant of Java.

SLAW's adaptive task scheduling is motivated by the study of two common scheduling policies in a work-stealing scheduler, specifically, the *work-first* and the *help-first* policy. Both policies exhibit limitations in performance and resource usage in different situations. The variances make it hard to determine the best policy a

priori. SLAW addresses these limitations by supporting both policies simultaneously and selecting policies adaptively on a per-task basis at runtime. Our results show that SLAW achieves 0.98× to 9.2× speedup over the help-first scheduler and 0.97× to 4.5× speedup over the work-first scheduler. Further, for large irregular parallel computations, SLAW supports data sizes and achieves performance that cannot be delivered by the use of any single fixed policy.

SLAW's locality-aware scheduling framework aims to overcome the cache unfriendliness of work-stealing due to randomized stealing. The SLAW scheduler is designed for programming models where locality hints are provided to the runtime by the programmer or compiler. Our results show that locality-aware scheduling can improve performance by increasing temporal data reuse for iterative data-parallel applications.

Acknowledgments

I owe my deepest gratitude to my advisor, Prof. Vivek Sarkar, for his guidance and support. He leads the Habanero Multicore Software Project with enthusiasm and is always patient in advising students. It is my great honor and pleasure to work with him. I would like to thank my thesis committee members, Prof. Keith Cooper, Dr. Bill Scherer and Prof. Lin Zhong for their insightful comments on this work.

I am grateful to all members of the Habanero team for the collaboration, helpful discussions and valuable feedback related to this thesis. Especially, I would like to thank Jisheng Zhao and Raghavan Raman for providing the compiler support. This work will not be possible without you. I would like to thank Rajikshore Barik for some very useful discussions that shaped many aspects of this work. Yonghong Yan, Jisheng Zhao and I had a heated and yet productive discussion on the design of the Hierarchical Places Tree model.

My sincere thanks go to Vijay A. Saraswat and Igor Peshansky for insightful discussions during their visit at Rice University.

This work was generously supported by the National Science Foundation under the HECURA program, award number CCF-0833166, IBM Open Collaborative Faculty Award and IBM CAS PhD Fellowship. I would like to thank Prof. Doug Lea for access to the UltraSPARC T2 SMP system used to obtain the experimental results in this thesis.

I am indebted to my parents, Zhongfang Guo and Limei Liu. They raised me and poured me with love unconditionally and have always been supporting and encouraging me during my PhD journey. Words are not enough to express my love and gratitude to you.

Last but not least, I would like to thank all my brothers and sisters in Rice Chinese Christian Fellowship (RCCF). Thanks for your prayers and support. I will cherish all moments we shared. Special thanks to Lily Lam for shepherding RCCF.

Contents

Illustrations

Tables

Chapter 1

Introduction

For decades, the computer industry has been delivering performance through *parallel computing*, a form of computation in which multiple calculations are performed simultaneously. Early computer systems in the 1970's exploited *bit-level parallelism* that resulted from simultaneously computing multiple bits in a processor subword or word during the execution of a single instruction. From the mid-1980's to 1990's, computer architecture designs are dominated by *instruction-level parallelism* (ILP), a form of parallelism in which multiple instructions are executed simultaneously. For example, modern processors have multi-stage instruction pipelines so that multiple instructions at different stages can all make progress at one CPU cycle. Superscalar processors can issue multiple instructions per cycle. Other micro-architectural ILP techniques include out-of-order execution, speculative execution and branch prediction. In order to support these ILP techniques, the processor logic is becoming increasingly sophisticated. For example, the Pentium 4 processor contain a 35-stage instruction pipeline, and a large portion of the silicon area is consumed by algorithms to enable or increase ILP [92, 86].

Gordon Moore predicted in 1965 that the number of transistors that can be placed inexpensively on an integrated circuit would double approximately every two years [66]. Historically, with the help of techniques that increase instruction-level parallelism, processor manufacturers were able to deliver generations of processors with a doubling of clock rate as the number of transistors doubled. Sequential programming languages and compilers were able to convert these hardware improvements to performance and productivity improvements without requiring programmers to

change their sequential code.

However, an exponential increase in clock rate is no longer sustainable due to power dissipation limits. The trend in the last few years has made it clear that the processor makers are now committed to multi-core chip designs. Nowadays, almost all computers, from high performance computers, to departmental and personal computers, and even embedded processors, are being built with multi-core chips. The number of cores per chip is increasing, while there is little or no increase in the clock speed per core. This parallelism trend poses a huge and urgent challenge on computer software because programs must be written or transformed into a multi-threaded form to take full advantage of future hardware advances.

It would be ideal to have the compiler automatically convert sequential code to multi-threaded code to take advantage of multi-core hardwares. Despite huge research effort in the 80's and 90's, such techniques, known as automatic parallelization, have had only limited success in specific situations such as regular loops and array accesses [7]. Fully automatic parallelization of sequential programs remains an open unsolved problem. Since no automatic parallelization solution is in sight, the search for *efficient and productive parallel programming models* for software developers has taken on a new urgency.

Parallelism programming models define how parallelism is expressed by programmers. Data parallelism and task parallelism are two common kinds of parallel programming models that express the parallelism from the data and the computation perspective respectively. Data parallelism focuses on data distribution across parallel computing nodes [25]. Many scientific applications contain loops that access large scale of data and such programs are suitable for data parallelism programming models. Google's MapReduce programming model [27] is a data parallel programming model especially designed for large scale data processing.

In contrast to data parallelism, task parallelism, also called function parallelism, expresses parallelism from the perspective of computations. In task parallelism,

programmers focus on decomposing the problem into sub-computations, or tasks, that can run in parallel. Entirely different computations can perform on either the same or different sets of data. How computations are actually scheduled and executed on the underlying architecture is the responsibility of the compiler and runtime systems. This separation of concerns between task decomposition and scheduling provides productivity, as shown in recent studies on different parallel programming models [32].

Task parallelism is considered a prerequisite for the productivity of generally parallel programming. The three programming languages developed as part of the DARPA High-Productivity-Computer-Systems (HPCS) project (Chapel [24], Fortress [65], X10 [20]) are all task-parallel languages. The past decade sees a fast increase in popularity of task-parallel systems. Among them are Java 5 Concurrency Library, Java Fork-Join Framework [57], Intel Thread Building Blocks [48], Microsoft .NET Task Parallel Library [58], Cilk [39], and etc. The task concept is also introduced into OpenMP since version 3.0 [70] which is traditionally designed for data parallelism of loops.

1.1 Task-Parallel Runtime Scheduler

A runtime serves as the environment in which computations are performed. It consists of the whole operating systems, including sub-systems such as memory management, thread/process management, and I/O management. A runtime usually appears in the form of libraries. A program interacts with the environment through runtime library calls dynamically. A program can only be executed in the compatible runtime environment and must interact with the runtime in a correct way, or a runtime error may occur. The runtime, on the other hand, is responsible for managing the resources and performing accordingly to the library calls. In the C programming language, for example, a program requests and free memory through `malloc` and `free` runtime library calls of the standard runtime memory management system. The runtime is

responsible for effectively managing the available memory in the system, and the program is responsible for using the runtime calls correctly. Errors such as memory leak and dangling pointers are considered fault of the program instead of the runtime.

A parallel runtime is a runtime where computations may be run in parallel. Running computations in parallel poses new challenges that demand the redesign of almost all parts of the runtime systems [11, 59]. In task parallelism, the task-parallel runtime scheduler is responsible for scheduling computations among threads. These threads, often called *workers*, are OS threads controlled by the operating systems.

Compared to the traditional programming model known as Single-Program-Multiple-Data (SPMD), scheduling computations among threads is a problem new to task parallelism. In SPMD model, the computation of each thread is specified by the programmer; In task parallelism, however, the separation of concerns between task decomposition and scheduling puts the burden on the task-parallel runtime schedulers to decide when to switch tasks and which task to execute for each worker. Though theoretically correct, creating one thread per task is not practical for performance concerns. For problems in which the number of tasks created is exponential to the input size, this approach will significantly over-subscribe the processors, leading to poor performance.

This dissertation focuses on the design and implementation of task-parallel runtime schedulers, especially the scalability, the overhead, and the resource usage bound under various kinds of task-parallel programs.

1.2 Work-stealing scheduler

A typical task-parallel system consists of a pool of workers, the number of which is decided by the number of underlying computing nodes. *Work sharing* and *work stealing* are two scheduling paradigms used to address the problem of scheduling multithreaded computations among the workers.

In work-sharing, whenever a new task is generated, the scheduler works *eagerly*

to re-distribute tasks through a shared task pool. In work-stealing, each worker maintains its own pool (queue) of tasks and the underutilized workers take the initiative to *steal* work from other busy workers. The worker that creates the new task pays a small overhead to enable stealing.

Compared to work-sharing, work-stealing schedulers have the following advantages. First, work-stealing is more efficient: the busy worker pays only a small overhead both to enable stealing on task creation and to execute tasks that are not stolen. The bulk of the overhead is shouldered by underutilized workers whose CPUs are idle anyway. Second, work-stealing is more scalable than work-sharing: in work-sharing, there is contention on a single shared task pool when pushing and popping tasks. This contention is distributed in work-stealing [29]. Third, it has been shown that work-stealing with certain policies can schedule tasks using bounded memory resource [15].

Although work-stealing has many advantages over work-sharing, the implementation of a work-stealing system is more complicated than that of a work-sharing scheduler. Implementation techniques for work-stealing systems has received a lot of attention since the advent of the Cilk work-stealing runtime developed by MIT [39].

Scheduling Policy

In work-stealing, the task scheduling policy determines the order in which tasks are executed. *Work-first* and *help-first* are two commonly used task scheduling policies used when spawning a task. Under the work-first policy, the worker will execute the spawned task eagerly, i.e., th worker *first works* on the spawned task. Under the help-first policy, the worker will defer the execution of the spawned task and instead execute the spawned task by continue execution on the parent task, i.e., the worker *first* asks for *help* from the other workers for executing the spawned tasks. This thesis shows that the work-first and help-first policies have different stack and memory bounds and also exhibit scalability limitations in different scenarios.

Locality

Locality is the phenomenon of the same value or related storage locations being frequently accessed by the same CPU. Modern computer architectures are built with multiple memory hierarchies, such as registers, L1, L2, L3 cache etc, in order to exploit locality at different levels. It is much more expensive to access data that lie further from the processor in the memory hierarchy than to access data that reside closer to the processor. As the memory hierarchy grows deeper, the relative gap in access time increases by orders of magnitude.

In task parallelism, although the programmer does not express the parallelism from the data perspective, it is still important to exploit locality to achieve good performance. Past research has shown that significant speedup can be achieved by making the work-stealing schedulers aware of the data locality of the tasks being scheduled [3]. This thesis presents a locality-aware scheduling frame and show examples of using the framework to improve the data locality in task parallelism.

1.3 Thesis Statement

Our thesis statement is as follows:

Work-stealing schedulers with policy adaptation and locality-awareness can provide a scalable and robust runtime foundation for dynamic task parallelism on multi-core systems.

1.4 Research Contributions

This dissertation makes the following contributions:

- A new work-stealing runtime system called SLAW for the Habanero-Java programming language, which is a Java-based task-parallel language. SLAW stands for Scalable Locality-aware Adaptive Work-stealing. SLAW has the compiler support to ensure the runtime APIs are always called correctly.

- A new work-stealing scheduling framework that support both work-first and help-first policies in async-finish parallelism.
- A non-blocking work-stealing deque implementation for garbage collected runtime systems like Java.
- A study of both pros and cons of different task scheduling policies in different applications considering both performance and resource usage bound.
- An adaptive work-stealing scheduling algorithm that can obtain the best of different scheduling policies with little or no additional overhead.
- A locality-aware work-stealing framework for programmers or compilers to exploit data locality (affinity) among tasks.

1.5 Thesis Organization

The rest of this thesis is organized as the following:

- Chapter 2 introduces the necessary background, definitions, notations and concepts used in the thesis. The Cilk language and runtime as well as the basic work-stealing implementation is discussed in this chapter.
- Chapter 3 presents the Habanero-Java programming language, which produces a class of computations characterized by *async-finish parallelism*. The properties of the async-finish parallelism is discussed in this chapter.
- Chapter 4 presents the implementation of SLAW including SLAW's task synchronization protocol and work-stealing deque extension.
- Chapter 5 presents an evaluation of task scheduling policies, the adaptive task scheduling algorithms, its theoretical bounds and the experimental results of locality-oblivious scheduling.

- Chapter 6 presents SLAW's locality-aware scheduling and examples to use the framework to improve data locality in task parallelism.
- Chapter 7 discusses the related work by comparing SLAW to other work-stealing systems.
- Chapter 8 concludes the thesis.

Chapter 2

Background

This chapter introduces the necessary background, definitions, notations and concepts used in the thesis.

2.1 Parallel Programming Models

Parallel programming models define how parallelism is expressed by programmers and how applications are matched to underlying parallel systems. Parallel programming models are judged by simplicity, expressibility and ability to deliver performance. The ultimate goal is to provide a simple and expressive parallel programming model without compromising much on performance.

Research on parallel programming models has a long history. Lots of parallel programming models have been proposed. In this section, we discuss the address space of parallel programming models, the traditional single-program-multiple-data (SPMD) model and the task parallelism model that has become increasingly popular in the past decade.

2.1.1 Address Space

In parallel programming, the address space defines how data are referred to. Two common address space models are the distributed memory model and the global address space model. In the distributed memory model, each data entity belongs to exactly one processor and can only be addressed by that processor. Access to remote data must be completed explicitly through communication. The distributed memory model is a natural match for parallel systems in which each processor has its

own private memory. These include most supercomputer clusters and heterogeneous accelerators such as GPUs. The advantage of the distributed memory model is that it forces programmers to think about data distribution and communication. As a result, it is more likely to produce a scalable program. The major disadvantages of distributed memory model are lack of simplicity and productivity as most sequential programmers are not used to the distributed memory model.

The global address space (GAS) model offers a single address space in which all data can be found. It is a natural match for parallel systems with shared memory, such as symmetric multi-processors (SMP). This model is a natural extension of commonly used sequential programming models. One drawback of the pure global address model is the lack of locality exploitation, which is required to achieve good performance on many modern architectures.

The partitioned global address space (PGAS) model aims to combine the productivity of the global address space model with the performance of the distributed memory model. PGAS assumes a global memory address space that is logically partitioned into portions. Each portion is local to one processor. Each data entity logically resides in one portion and has affinity with the processor that is local to the portion.

2.1.2 SPMD Model

Single-Programming-Multiple-Data (SPMD) is a commonly used traditional parallel programming model in which the same program is launched on multiple processors while each processor operates on its own portion of data. The term was first coined by Darema et al. [26] In the SPMD model, each thread has a unique identifier to distinguish itself with other threads. The thread identifier maps the computations of the application to the threads.

Message Passing Interface (MPI) is the de facto standard for communications between threads in parallel systems. MPI is based on the SPMD model and the

distributed memory model. Programmers using MPI write a single program that is launched on multiple processes while each process has its own separate address space. Processes communicate and synchronize with each other explicitly through a standard message passing interface. MPI is widely used in computer clusters and supercomputers due their high performance and portability. However, developing MPI programs can be time-consuming and error-prone. For example, when the programmer has the burden of managing all communication and synchronization in MPI, the program is prone to deadlock.

OpenMP is a popular standard for shared memory parallel programs. OpenMP is based the global address space model and uses the SPMD model for parallel regions. OpenMP programmers typically start with a sequential program and enable parallelism by adding program directives. Prior to OpenMP 3.0, OpenMP primarily focused on data parallelism of loops. Since version 3.0, OpenMP adopts the task concept and is a mix of SPMD and task parallelism.

Some PGAS languages are also based on the SPMD model. For example, UPC [33] and CAF [68] are PGAS languages that extend C and Fortran respectively with SPMD and a partitioned global address space.

Load Balancing in SPMD

During the execution of parallel programs, threads often synchronize with each other through synchronization points such as barriers or blocking operations. Load imbalance happens when faster threads reach the synchronization pointer before slower threads, and the faster threads have to wait for the slower threads. Load imbalance slows down the overall performance.

In modern applications and architectures, many factors may cause load imbalance, including asymmetric processor speed, processor over-subscription and unevenly divided work between synchronization points. For some irregular applications, it is hard to evenly divide and map the work to threads.

Load balancing is especially challenging SPMD programs in which the mapping between computations and threads are specified by the programmer, and faster threads may not be able to help with the computations of slower threads. Load balancing for SPMD has been a topic of lots of research [59, 47, 16, 35]. For OpenMP parallel loops, the programmer can specify the chunk size of the loop and how these chunks should be scheduled. Hofmeyr et al. designed a tool to performance load balancing of threads at the OS level in cases of processor over-subscription [46].

2.1.3 Task Parallelism

Task parallelism is a kind of parallel programming model in which programmers focus on decomposing the problem into sub-computations, called tasks, that can run in parallel. The task can be any piece of code, and the amount of work contained varies from task to task. Programmers creating tasks without worrying about how these tasks are mapped to the underlying threads. From the programmer's view, task scheduling and load balancing are automatically handled by the runtime. The task-parallel runtime is responsible for scheduling tasks and migrating task from busy threads to idle threads for load balancing.

Studies have shown that the task parallel model is more productive than the SPMD model [32]. Task parallelism is becoming increasingly popular in the past decade, and both task-parallel languages and libraries have been introduced. All three programming languages supported by DARPAs High Productivity Computer Systems (HPCS) project, which aims for developing a new generation of economically viable high productivity computing systems, are dynamic task parallel language (X10, Chapel, Fortress). StackThreads/MP and Intel Thread Building Block (TBB) are C-based task parallel libraries. Doug Lea's Java ForkJoin Framework [57] is a Java-based task parallel library.

2.2 Task Scheduling

This section presents the task scheduling model assumed in this thesis and discuss task scheduling paradigms.

2.2.1 Scheduling Model

A multithreaded computation can be modeled as a *computation dag* (directed acyclic graph) of dynamic instruction instances connected by dependency edges as in [15, 5]. The instructions within a task are connected by *continue edges*, which represent the sequential ordering of instructions.

During the execution of a program, a task may create, or spawn a child task so that the spawned child task may run in parallel with the parent task. In this case, a *spawn edge* is used to represent the dependency from the spawn instruction in the parent task to the first instruction of the child task. Each task except the root task has exactly one parent task. All tasks are connected into a *spawn tree* by spawn edges.

Besides, continue edges and spawn edges, *join edges* are used to represent dependencies that may cause a task to *stall* at some instruction, waiting for the completion of other instructions.

An example of a computation dag is shown in Figure 2.1. There are 6 tasks in the computation dag, Γ_1 to Γ_6. The spawn tree is shown in Figure 2.2.

Continuation is a term originally used to represent the rest of a computation after some point. The continuation of a task γ after some instruction v consists of all instructions that can be reached by continue edges in task γ from instruction v. For example, the continuation after v_3 consists of instructions $v_6, v_9, v_{10}, v_{11} and v_{15}$. The term continuation is closely related to the term execution context, which represents the information need to resume execution of a computation. The execution context of a computation at one point is all the information needed to resume the execution of the continuation after that point. Without ambiguity, these terms are often used interchangeably.

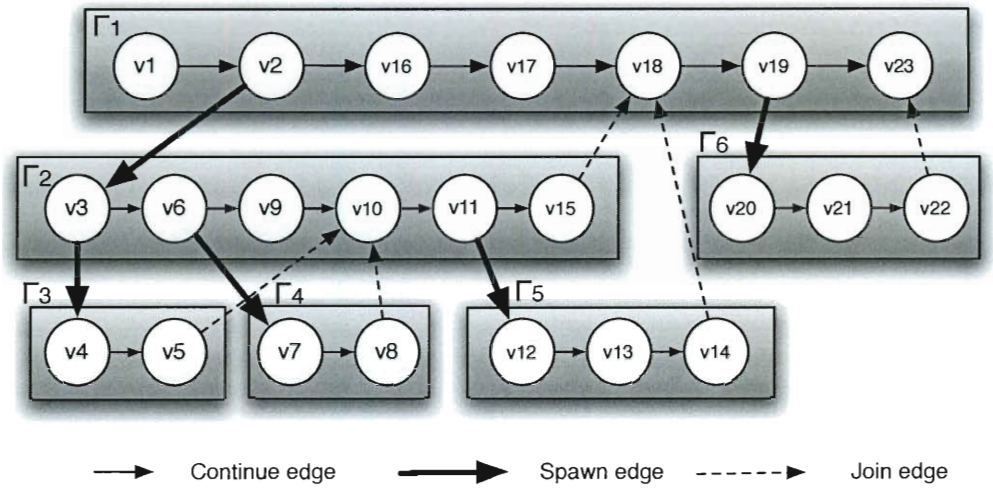


Figure 2.1 : Computation Dag

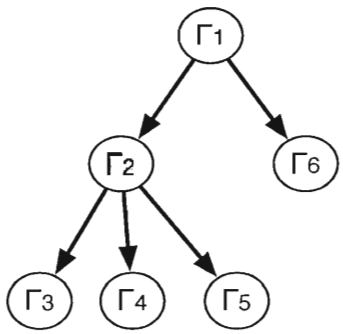


Figure 2.2 : Task Spawn Tree

Given a computation dag, if every join edge goes from a task to its spawn tree ancestor, the computation is called a *strict* computation. If every join edge goes from a task to its spawn tree parent, the computation is called a *fully-strict* computation [15]. If a computation is strict and every join edge goes from the *last* instruction of a task to its spawn tree ancestor, the computation is called *terminally-strict* [5].

A P-processor execution schedule of a multithreaded computation dag determines which processor of a P-processor parallel computer executes which instruction at each time step. A legal execution schedule must observe all dependencies, and, at any given step, each processor may execute at most one instruction.

Each task allocates a chunk of memory, called its activation frame, when it starts execution. This activation frame is used to store the local state of the task. The activation frame may be deallocated after the last instruction of the task is executed. In the serial depth-first execution of a multithreaded program, activation frames represent the activation records on the call stack. The total maximum amount of the memory used in the serial depth-first execution of the computation is denoted as S_1.

Given a computation dag, we use T_1 to denote the total execution time of all instructions in the computation dag and T_∞ to denote the execution time spent on the critical path. T_1 is also called the *work* of the computation and is the execution time on a single processor. T_∞ is also called the *span* of the computation and is the minimal execution time on an unbounded number of processors. The parallelism $\bar{P}$ of the computation is defined as $\bar{P} = T_1/T_\infty$.

Note that the computation dag should be studied in an *a posteriori* fashion. The computation dag of a multithreaded program often depends factors that are not known until runtime.

2.2.2 Scheduling Paradigms

Work-sharing and work-stealing are two task scheduling paradigms for task parallelism. In work-sharing, when a new task is created, the creator works eagerly to re-distribute the new task. The task re-distribution in work-sharing is usually implemented by a centralized task pool. As shown in Figure 2.3, new tasks are inserted to the task pool by busy workers (e.g., w_1, w_2), while idle workers (e.g., w_3, w_4) are polling tasks from the pool. In a multithreaded environment, all accesses to the task pool need to be synchronized. The centralized task pool can become a scalability bottleneck when the number of workers increases, or when many fine-grain tasks are created. The X10 v1.5 runtime implements the work-sharing paradigm using `java.util.concurrent` ThreadPool Executor class [10].

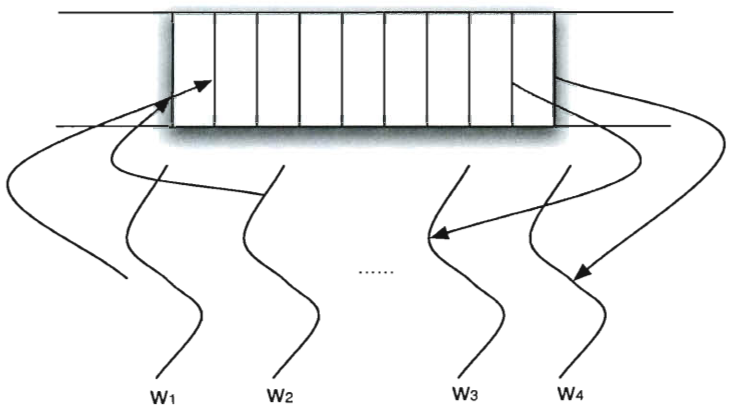


Figure 2.3 : Work-sharing Scheduling Paradigm

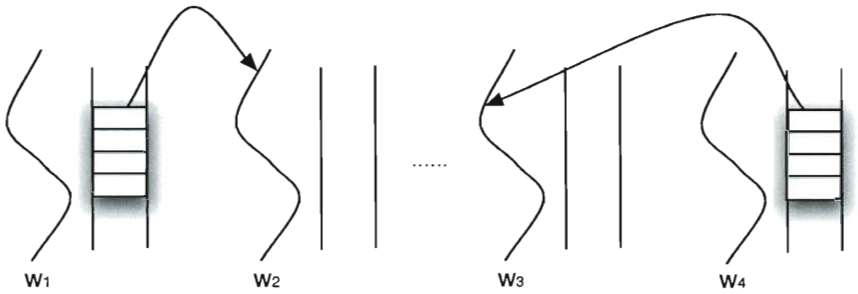


Figure 2.4 : Work-stealing Scheduling Paradigm

In work-stealing, however, the underutilized workers take the initiative to steal work from other busy workers. The busy worker only pays a small overhead to enable the stealing. Work-stealing is CPU-efficient because the major overhead is incurred by the underutilized idle workers for which the CPU cycle is wasted anyway. As shown in Figure 2.4, work-stealing is implemented through distributed task pools.

The idea of work-stealing dates back at least to Burton and Sleep's research in 1980s on execution models for functional programs on large number of computing elements [18] and Halstead's Multilisp implementation [42]. In 1990s, Blumofe and Leiserson presented a work-stealing scheduling algorithm and provided time, space and communication bounds for the parallel execution using the work-stealing algorithm on the class of fully-strict computation [15]. Their work justified the folk wis-

dom that work-stealing is more efficient than work-sharing. Blumofe's work-stealing algorithm is employed in the runtime of the C-based task parallel language called Cilk [39]. Blumofe and Leiserson's pioneer work in work-stealing and Cilk greatly inspire the design and implementation of Habanero-Java's work-stealing runtime.

The implementation of work-stealing is critical in order to achieve good performance and bounded resource usage. Section 2.3 describes the background of the work-stealing implementation. In particular, Section 2.3.3 discusses the Cilk implementation.

2.3 Work-stealing Implementation

This section describes the background of the work-stealing implementation. We first describe some basic concurrency constructs. Then we discuss two common forms of memory allocation, stack and heap allocation. Finally, we describe the implementation of work-stealing systems using Cilk as an example. In particular, we discuss the resource bound of work-stealing, the work-stealing deque implementation and the work-stealing compiler support.

2.3.1 Basic multithreaded programming

This section describes three basic concurrency structures in multithreaded programming as well as the the Java 5 memory model.

Lock

A lock is a synchronization mechanism for enforcing exclusive access to a resource in an environment where there are many threads of execution. Locks are one way of resolving concurrency conflicts.

Locks are supported in most modern operating systems; however, locks have known performance disadvantages: First, locks may cause the thread that tries to acquire the lock be blocked by the operation system until the lock is released. In the

worst case, the computation may be restricted to a single thread execution at any given time. Second, locking adds overhead, even for situations when the chance of conflict is known to be small.

Atomic Operations and Lock-free Algorithms

Many non-blocking algorithms, or lock-free algorithms, have been proposed to overcome the performance disadvantages of locks [63, 84, 87, 77]. Non-blocking algorithms relies on *atomic* operations. For example, *compare-and-swap* operation atomically compares the content of a memory location to the given value and, if they are the same, modifies the content of that memory location to the new value given. The return result of the *compare-and-swap* operation must indicate whether it performed the substitution or not.

Compare-and-swap operation is now supported in hardware by many computer architectures and results in lower overhead than locks if the chance of conflict is small. Besides, *compare-and-swap* operation will not cause the thread to be blocked by the operation system.

Memory Barrier

Memory barriers are instructions that cause a processor to enforce ordering constraints on memory operations issued before and after the barrier instruction. A barrier can also be a high level programming language statement that prevents the compiler from reordering certain operations over the barrier statement during optimization passes.

Different classes of barrier exist and may apply to specific sets of operations. A load-load barrier prevents the reordering of two load instructions before and after the barrier. A store-store barrier prevents the reordering of two store instructions before and after the barrier. A load-store barrier prevents the reordering of load instructions before the barrier with the store instruction after the barrier. A store-

load barrier prevents the reordering of store instructions before the barrier with the load instruction after the barrier.

Memory barriers are used in many concurrency protocols involving parallel threads to ensure proper ordering of instructions [28, 55, 71]. Memory barrier is also used to implement the memory model.

Java 5 memory model

In parallel computing, the memory model specifies, for each read operation, what is the set of store operations whose result can be returned [60, 4]. *Sequential consistency* is the memory model that most programmers can naturally reason about [54]. It requires that the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the program order. The simplicity of the sequential consistency model is achieved at the cost of performance. Most modern architectures support consistency models that are weaker than sequential consistency.

The current Java memory model is a result of JSR 133 [73], which fixed serious flaws in the memory model before Java 5 [74]. In the new Java 5 memory model, all synchronization actions (e.g., read or write to a volatile variable) form a partial order called *happens-before* order. This happens-before order subsumes the program order: if one action occurs before another in the program order in one thread, it will occur before the other in the happens-before order. The happens-before order for synchronization actions from different threads can vary from execution to execution. For a particular execution, one read is allowed to return the value of a write if that write is the last write to that variable before the read along some path in the happens-before order, or if the write is not ordered with respect to that read in the happens-before order. Atomic variable is implemented through a volatile variable and all access to an atomic variable is considered as access to volatile variable.

The example in Figure 2.5 illustrates how the programmer can reason about the

value that returns from the read operation. In the program, variable `done` is volatile and `save` is not. One thread executes function `done()` and another thread executes function `check()`. In this example, when the read of `done` returns true (line 10) in `check()` and the branch is taken, the new Java memory model assures that any read of `save` in the branch will return 5.

The reasoning is as follows: The write to `save` precedes the write to `done` in program order. So there is a happens-before order from the write to `save` (line 5), to the write to the volatile variable `done` (line 6). Similarly, the read of the volatile variable `done` (line 10) happens-before the read of variable `save` in the branch (line 11). If the branch is taken, this implies the read of `done` at line 10 returns true, which further implies that line 6 happens-before line 10 in this execution. By the transitivity of the happens-before partial ordering, the write to the `save` at line 5 happens-before the read of the variable `save` in the branch. Thus the read of `save` in the branch must return 5. Under the old Java memory model before Java 5, the compiler may reorder the write to `save` at line 5 after the line 6 even `done` is declared *volatile*.

The implementation of the work-stealing runtime in this thesis assumes on the Java 5 memory model.

2.3.2 Stack and Heap Allocation

Memory usage is an important consideration in task scheduling. Stack and heap allocation are two forms of memory allocation at runtime. If the memory usage is not managed carefully, the parallel execution of the program may be terminated prematurely due to either stack overflow or heap overflow.

In most modern computer architectures, each process or thread has a reserved memory space referred to as its stack (also known as execution stack, runtime stack or call stack). The stack is composed of activation records. Each activation record corresponds to a call to a function: a new activation record is pushed onto the stack

```
1  class VolatileExample {
2    int save = 0;
3    volatile boolean done = false;
4    public void done() {
5      save = 5;
6      done = true;
7    }
8
9    public void check() {
10     if (done) {
11       // guarantees that save = 5
12     }
13   }
14 }
```

Figure 2.5 : Sample use of volatile variables in Java 5

when a function is called and the topmost activation record is popped when the function returns. The content of the activation record is machine dependent and typically includes return address to the caller, parameter values and local variables.

Stack allocation is also called automatic allocation as the memory is automatically allocated and freed as the function is called and returns.

The *execution context* of a thread refers to all information need to be saved when a thread's execution is interrupted and will resume later. The call stack of the thread contains the value of local variables and is an important part of the thread's execution context.

Besides the stack allocation described above, heap allocation is another form of memory allocation during the runtime of a program. Heap allocation is also called dynamic allocation. The life time of the dynamically allocated memory exists until it is freed by the programmer or collected by a garbage collector at runtime. The pool of unused memory structured for dynamic allocation is called the heap. In the C programming language, heap allocation is realized using the `malloc` function in the

standard library.

In multithreaded programming, each system-level thread has a *separate* stack, the size of which is fixed by the operating system when the thread is created; however, although the implementation of the heap allocation varies, each thread should be able to dynamically request more memory from other threads or the operation system.

The major advantage of stack allocation over heap allocation is its simple and efficient implementation. Different CPU architectures, operating systems, and programming languages may use different calling conventions; however, most calling conventions can be implemented with a few assembly instructions. For example, the x86 processors family has hardware support for manipulating the stack of the executing thread.

Due to the dynamic nature of heap allocation, the implementation of dynamic memory allocation is not simple. Most research focuses on two problems: the inefficient space usage caused by fragmentation and the concern of scalability when multiple threads are allocating and freeing objects frequently. Although, we have seen great progress towards space-efficient and scalable dynamic memory allocators, the bottom line is that the dynamic allocation is still much slower than the stack allocation. In order to reduce the object allocation overhead, some compilers allocate objects on stack instead of on heap when legal to do so.

Despite the advantages, stack allocation also comes with limitations. First, some objects have to be heap allocated if they need a longer lifetime than that of the function that creates them. Second, the maximum size of the stack is fixed when the thread is created by the operating system, and can be as small as several Kilobytes or Megabytes, while the size of the heap size is much larger. Using more stack than is available will result in a crash due to *stack overflow*. We use the term, *stack pressure*, to indicate how close a thread is to incurring stack overflow. The stack pressure is increased when a function is called and decreased when the function returns. The stack pressure can also be reduced by special forms of return operations such as a

`longjmp` instruction or throwing an exception. Recursive programs and programs that make a lot of function calls are especially prone to stack overflow. One optimization to reduce stack pressure is tail-call optimization. A tail call in a function `f()` is a function call which is followed by a return to the caller of `f()`. Tail call optimization removes the call stack manipulation of a tail call, and replaces the call by a jump to the callee. Tail call optimization is used in many functional programming languages which have higher stack pressure than traditional imperative language. In Scheme, tail call optimization is mandatory.

2.3.3 Cilk's work-stealing runtime

Cilk is a C-based task parallel programming language. In Cilk, the keyword `spawn` indicates the function call that follows can safely run in parallel with other executing code. The keyword `sync` indicates that execution of the current procedure cannot proceed until all previously spawned functions have completed and returned their results to the parent frame. There is an implicit `sync` before the end of each function. As a result, all computations produced by Cilk are fully-strict.

Theoretical bounds

Cilk's work-stealing runtime [39] is based on Blumofe's work-stealing algorithm for fully-strict computation [15]. Theoretical analysis shows that the work-stealing algorithm has the following properties:

time efficient: the expected time to execute a fully strict computation on P processors is $T_1/P + O(T_\infty)$. This suggests that if there is abundant parallelism in the computation, the scheduler can achieve almost linear speedup on P processors.

space efficient: the space required by the execution is at most $S_1 P$, i.e., P times the space requirement of the depth-first execution. The space includes all memory allocations and does not distinguish between stack and heap allocation.

communication efficient: the expected total communication between processors of the algorithm is at most $O(PT_\infty(1 + n_d)S_{max})$, where S_{max} is the size of the largest activation frame and n_d is the maximum number of times that any thread synchronize with its parent. The communication can be considered as the product of the expected number of steals $O(PT_\infty(1 + n_d))$ and the amount of data migrated during each steal S_{max}. The number of steals is expected to increase with the growth of the number of processors and the critical path.

These theoretical properties justified the folk wisdom that work-stealing is more efficient than work-sharing.

Work-first principle and its assumptions

Let T_S be the execution time of the serial C-equivalent (also called the C-elision) of the Cilk program. Cilk's compiler and runtime tend to remove overhead away from the work of the computation (T_1) and minimizes the work overhead T_1/T_S. This is called the work-first principle, which pervades many designs of the Cilk's work-stealing runtime, such as the *THE protocol* of deque operations described in Section 2.3.4 and the two-clone compilation strategy described in Section 2.3.5. The work-first principle allows Cilk to significantly reduce the cost of spawning a parallel task.

The work-first principle is justified based on three assumptions:

1. First, the Cilk's work-stealing runtime runs in practice according to the theoretical analysis described above.

2. Second, parallel slackness exists in the computation, i.e., $\bar{P}/P$ is sufficiently large. Thus, the number of steals expected is relatively small.

3. Third, the execution time of the C-elision of the Cilk program T_S can be measured. In other words, it assumes the serial depth-first execution must complete execution successfully within the resource limit.

Although the design of the Habanero-Java work-stealing runtime largely follows the work-first principle, there are cases in which some extra code is introduced in T_1. Our thesis is that these extra code introduces only a small amount of overhead, but significantly improve the scalability and robustness of the runtime when some assumptions of the work-first principle do not hold.

The work-first principle shall not be confused with the work-first task scheduling policy described in this thesis. The work-first principle refers to the principle that removes overhead from the work of the computation (T_1) . The work-first task scheduling policy determines the order in which the tasks are executed. The work-first task scheduling policy is also called the depth-first scheduling policy in some literature.

2.3.4 Work-stealing deque

Most work-stealing runtimes use double-end queues (deques) to store tasks. The performance of the deques is crucial to a work-stealing runtime and has attracted much research [8, 44, 21, 64].

In a typical work-stealing runtime, every worker thread has a local double-ended queue (deque) to store tasks. When a new task is created, it is *pushed* to the bottom-end of the local deque of the worker. When a worker is idle, it first attempts to get a new task by *popping* task from the bottom-end of the local deque, before making attempts to *steal* from the top-end of other threads' deques. Because there is only one worker, i.e., the owner of the deque, can push and pop the deque from the bottom-end, there is no contention between the operations at the bottom-end of the deque; The conflicts between workers are caused by the steal operations, and thus called *steal conflict*. Multiple workers can steal from the top-end of the deque, so there can be steal conflict between the thieves. Steal conflict can also happens between the thieves and the owner when there is only one task in the deque.

Cilk uses the *THE protocol* to manage deque operations as illustrated in Figure 2.6.

```
1  push () {
2        bottom++;
3   }
4
5   pop() {
6        bottom--;
7        if (top > bottom) {
8           bottom++;
9           lock(deque);
10          bottom--;
11         if (top > bottom) {
12            bottom++;
13            unlock(deque);
14            return FAILURE;
15         }
16       unlock(deque);
17    }
18    return SUCCESS;
19 }
20
21  steal() {
22      lock(deque);
23      top++;
24      if (top > bottom) {
25         top--;
26         unlock(deque);
27         return FAILURE;
28      }
29      unlock(deque);
30      return SUCCESS;
31 }
```

Figure 2.6 : THE protocol (from [39] with variable renaming)

Under the 1-processor execution, the push and pop operations will contribute to the work overhead T_1/T_S, while the steal operation does not. So under the work-first principle, push and pop should be optimized even at the cost of an expensive steal operation. Under the THE protocol, push operation does not grab a lock. The pop operation does not grab a lock in most cases unless there can possibly be steal conflicts. The steal operation, on the other hand, always grabs a lock to resolve steal conflicts.

The pseudo-code in Figure 2.6 assumes sequential consistency. On most architectures, memory barriers are inserted in the code to maintain sequential consistency. Two store-load barriers are required between line 10 and 11 in the pop function and between line 23 and 24 in the steal function respectively. THE protocol is blocking because it uses lock to achieve mutual exclusion.

Some approaches have been proposed to improve the THE protocol.

Arora, Blumofe and Plaxton proposed a non-blocking deque algorithm for task scheduling. Their deque implementation is called ABP deque [8]. ABP deque uses of a pre-allocated fixed array and cannot properly handle the deque overflow situation. This problem is fixed by the Chase-Lev's deque implementation [21] described below.

Figure 2.7 to 2.11 shows the pseudo-code of Chase-Lev's work-stealing deque. Chase-Lev's work-stealing deque has a dynamic circular array and two index pointers: top and bottom. The circular array can grow dynamically. In order to prevent the growth from interfering with other operations, the new array is constructed in such a way that the index of the elements in the old circular array remain the same in the new circular array. Therefore, old index pointers can be used to refer elements in both old and new arrays.

Both index pointers are 64-bit long and monotonic. The monotonicity of the top pointer guarantees the object `o` retrieved at line 61 is indeed the top entry of the deque when the `cas` operation at line 62 returns true. Imagine the situation when the thief thread is suspended after line 61 and the deque is reconstructed before the

```
1  public class CircularWSDeque {
2      public final static Object Empty = new Object();
3      public final static Object Abort = new Object();
4      private final static int LogInitialSize = ... /* Log of
5                                                  the initial array size */;
6      private volatile long bottom = 0;
7      private volatile long top = 0;
8      private volatile CircularArray activeArray =
9                  new CircularArray(LogInitialSize);
10     private boolean casTop(long oldVal, long newVal) {
11         boolean preCond;
12         atomically {
13               preCond = (top==oldVal);
14               if (preCond)
15                   top=newVal;
16          }
17           return preCond;
18 }
```

Figure 2.7 : Pseudo code for CircularWSDeque class

thief thread resumes at line 62. Although the top pointer may return the same value at line 61 and 62, the top entry of the deque may not be the same as the deque is reconstructed. In parallel programming, this kind of problem is known as the *ABA problem.*

The work-stealing deque of Habanero-Java runtime is a modified version of Chase-Lev's dynamic circular work-stealing deque. The modification is to fix the memory leak problem of the Java implementation of Chase-Lev's algorithm. The implementation is described in Section 4.5.1.

```
19 public void pushBottom(Object o) {
20   long b = this.bottom;
21   long t = this.top;
22   CircularArray a = this.activeArray;
23   long size = b - t;
24   if (size >= a.size()-1) {
25     a = a.grow(b, t);
26     this.activeArray = a;
27   }
28   a.put(b, o);
29   bottom = b+1;
30 }
```

Figure 2.8 : Pseudo code of pushBottom operation

2.3.5 Language and Compiler Support

A program can only be executed in the compatible runtime environment and must interact with the runtime in a correct way, or a runtime error may occur. Some work-stealing runtimes deliver in the form as a library without compiler support [57, 58, 48]. The library approach is backward compatible: old programs still run correctly and programmers who wish to exploit parallelism can rewrite them using the new work-stealing library. However, the lack of compiler support reduces productivity and limits the full functionality of task scheduling. The productivity is reduced because programmers are given the burden to correctly interactive with the library when creating and synchronizing tasks. The functionality of task scheduling is limited because important information, the continuation of a task, is not available to the task scheduler without compiler support. It is awkward to make programmers explicitly specify the continuation of a task.

Dynamic task parallel languages simplify the effort to parallelize a sequential program and allow programmers to take full advantage of the runtime. The dynamic task parallel language is typical based on a sequential programming language to

```
31 class CircularArray {
32   private int log_size;
33   private Object[] segment;
34   CircularArray(int log_size) {
35     this.log_size = log_size;
36     this.segment = new Object[1<<this.log_size];
37   }
38   long size() {
39     return 1<<this.log_size;
40   }
41   Object get(long i) {
42     return this.segment[i%size()];
43   }
44   void put(long i, Object o) {
45     this.segment[i%size()] = o;
46   }
47   CircularArray grow(long b, long t) {
48     CircularArray a = new CircularArray(this.log_size+1);
49     for (long i=t; i<b; i++) {
50       a.put(i, this.get(i));
51     }
52     return a;
53   }
54 }
```

Figure 2.9 : Growth function of the circular deque

express the semantics of the program and has parallel extensions to express parallelism. For example, the Cilk programming language is C-based and uses **spawn** and **sync** to create and synchronize tasks. The compiler support is required to bridge the gap between a dynamic task parallel language with its runtime. In fact, the compiler and the runtime need to work together to support a new language or runtime feature [39, 89, 85, 75]. The compilation strategy of the Cilk compiler is shown below.

```
55 public Object steal() {
56   long t = this.top;
57   long b = this.bottom;
58   CircularArray a = this.activeArray;
59   long size = b - t;
60   if (size <= 0) return Empty;
61   Object o = a.get(t);
62   if (! casTop(t, t+1))
63      return Abort;
64   return o;
65 }
```

Figure 2.10 : Pseudo code of steal operation

The Cilk compiler generates two clones for each cilk function: the *fast clone* and the *slow clone*. The fast clone is always invoked on a spawn and is the clone executed in 1-thread execution. According Cilk's work-first principle, the fast clone is designed to bear as few overhead as possible. The slow clone is invoked when the thief steals a frame from a victim and then needs to resume execution from its appropriate continuation. The slow clone contains operations to restore execution context such as global and local variables etc.

Figure 2.12 shows the fast clone generated for the classical Fib example in Cilk. Upon a spawn, the *continuation* is saved in a frame which is pushed onto the worker's deque (line 20) so that other workers can steal it. *Continuation* represents the work in the current task after the spawn point. It contains the entry point (line 18) and the value of the necessary variables that are required to resume execution in the slow clone (line 19). After the continuation is pushed to the deque, the worker will execute the spawned task eagerly (line 21). Whenever the worker returns from a spawned task, it will first check if its deque is empty (line 22). If so, the worker aborts and becomes a thief (line 23). Otherwise, it pops the bottom-most frame and continue execution.

```
66 public Object popBottom() {
67   long b = this.bottom;
68   CircularArray a = this.activeArray;
69   b = b - 1;
70   this.bottom = b;
71   long t = this.top;
72   long size = b - t;
73   if (size < 0) {
74      bottom = t;
75      return Empty;
76   }
77   Object o = a.get(b);
78   if (size > 0)
79      return o;
80   if (! casTop(t, t+1))
81      o = Empty;
82    this.bottom = t+1;
83    return o;
84 }
```

Figure 2.11 : Pseudo code for CircularWSDeque class

2.4 Locality

Blumofe's work-stealing has both advantages and disadvantages in terms of cache locality. On one hand, Blumofe's work-stealing algorithm tends to execute tasks in the same order as if it were in the sequential execution, and it is believed that there is inherent data locality in the sequential execution [14, 67, 3]. On the another hand, the randomized stealing in Blumofe's work-stealing algorithm is cache-unfriendly. In particular, for iterative data-parallel programs, the randomized stealing prevents temporal data reuse between iterations. Acar et al. presents the lower and upper bound on the number of cache misses using Blumofe's work-stealing algorithm on hardware-controlled shared-memory machines [3].

To exploit data locality between tasks, Habanero-Java uses a PGAS address model and has a locality-aware scheduling framework. The details are presented in Chapter 6.

```
1  int cilk fib(int n) {
2        if (n<2) return n;
3        int x = spawn fib(n-1);
4        int y = spawn fib(n-2);
5        sync;
6        return x+y;
7  }
8
9  int fib(int n) {
10       fib_frame *f;
11        ...
12       if (n<2) {
13           ....
14           return n;
15       }
16       else {
17            int x,y;
18            f->entry = 1; // saving continuation (entry point)
19            f->n = n;  // saving continuation (n)
20            push(f);  // push continuation to deque
21            x = fib(n-1); // call fib normally
22            if (pop(x) == FAILURE) // check if the continuation is stolen
23                  return 0;
24            ....
25            ;
26            ....
27            return x+y;
28       }
29 }
```

Figure 2.12 : Fib's Cilk program and its compiler-generated fast clone (from [39] with slight modification)

Chapter 3

Habanero-Java and Async-Finish Parallelism

This chapter describes the subset of the Habanero-Java programming language that is supported by the work-stealing runtime and defines the async-finish parallelism model.

3.1 Habanero-Java

Habanero-Java (HJ) [1] is a Java-based dynamic task-parallel language derived from X10 v1.5 [20]. The full Habanero-Java language has a work-sharing runtime derived from X10 v1.5 runtime. The parallel constructs described in this chapter is the subset that are currently supported by a work-stealing runtime.

3.1.1 Asynchronous Task Creation: async

The `async` construct is used to create (fork/spawn) a statement as a new asynchronous task.

stmt ::= `async` [schedule(wf | hf | dyn)] [(place_expr)] ⟨*stmt*⟩;

This statement causes the parent task to create a new child task that executes ⟨*stmt*⟩. The parent task and the child task can run in parallel. An `async` statement can optionally include a *schedule* clause and/or a *place* clause [1]. The *schedule* clause can take one of three possible values: *wf*, *hf* or *dyn*, which correspond to the *work-first*, *help-first* and *dynamic* scheduling policies respectively. Task scheduling policies are discussed in Chapter 5. By default, task scheduling policies are selected dynamically

[1]The full Habanero-Java language also supports phaser registration clauses [79] in async statements

(*dyn*) on a per-task basis at the runtime. The *place* clause takes a place expression p, and serves as an affinity hint for the task. Places are described in Section 3.1.4.

3.1.2 Task Synchronization: finish

stmt ::= finish ⟨stmt⟩;

This statement causes the parent task to execute ⟨*stmt*⟩ and then wait until all subtasks created within ⟨*stmt*⟩ have terminated (including transitively spawned tasks). Operationally, each instruction executed in a task has a unique *immediately enclosing finish* instance/scope (IEF) [41]. Each dynamic instance of a `finish` statement can be viewed as being bracketed between matching instances of `startFinish` and `endFinish` instructions. An `endFinish` instruction for a dynamic finish F instance serves as a join synchronization for all tasks with $IEF = F$. The main function is enclosed by an implicit finish instance.

Figure 3.1 shows an example HJ code fragment and its computation dag. The first instruction of the main task serves as the *root* node of the dag (with no predecessors). Any instruction which spawns a new task will create a child node in the dag with a *spawn* edge connecting the `async` instruction to the first instruction of that child task. HJ tasks may wait on descendant tasks by executing a `finish` statement. We model these dependencies by introducing `startFinish` (l_2 in Figure 3.1) and `endFinish` (l_8 in Figure 3.1) nodes in the dag for each instance of a `finish` construct and then create *join* edges from the last instruction of each spawned task within the scope of `finish` to the corresponding `endFinish` instruction.

3.1.3 Atomicity: isolated

stmt ::= isolated ⟨stmt⟩;

The *isolated* construct is HJ's renaming of X10's *atomic* construct. As stated in [20], an atomic block in X10 is intended to be "executed as if in a single step during which all other concurrent tasks in the same place are suspended". This

```
l1 S0;
l2 finish { //startFinish
l3     async {
l4         S1;
l5         async {
l6             S2;}
l7         S3;}
l8 } //endFinish
l9 S4;
```

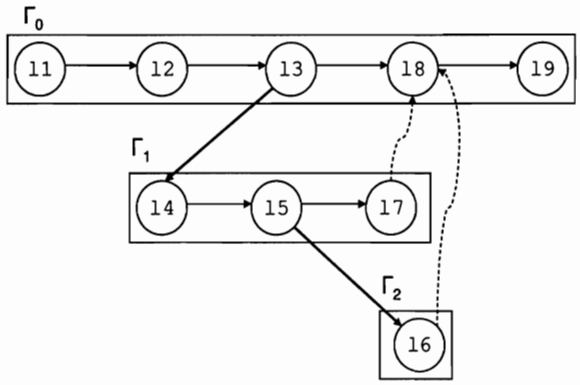


Figure 3.1 : HJ computation dag

definition implies *strong atomicity* semantics for the atomic construct. However, all X10 implementations that we are aware of are lock-based and do not enforce any mutual exclusion guarantees between computations within and outside an atomic block. As advocated in [56], we use the *isolated* keyword instead of *atomic* to make explicit the fact that the construct supports weak isolation rather than strong atomicity.

3.1.4 Places

Habanero-Java is based on the Partitioned-Global-Address-Space (PGAS) [93] address model. The global address space is partitioned into multiple *places*. The number of places is fixed when the program is launched; there is no construct to

create a new place. This is consistent with current programming models, such as MPI [81], UPC [33], and OpenMP [69], that require the number of processes/threads to be specified when an application is launched.

Unlike X10, current HJ only associates places with tasks not with data. Data locality is instead achieved indirectly by say assigning two tasks with the same data affinity to execute in the same place.

Each task initially has a place expression to represent its locality attribute. On hardware-controlled shared-memory machines, the locality attribute serves as a locality hint as the runtime is free to schedule the task on any worker. At runtime, a task can obtain a reference to the place of the executing worker by evaluating the place expression *here*. Upon spawning a child task in an `async` statement, the child task will derive its locality attribute from its parent's dynamic place, unless an place expression is specified.

3.1.5 Task scheduling points and restrictions

In Habanero-Java runtime, a worker can switch from one task to another task only at *task scheduling points* (also called task switching points). There is a task scheduling point (1) before and after each asynchronous task, and (2) after each `endFinish`.

Unlike OpenMP 3.0 which has task scheduling restrictions regarding tied tasks, Habanero-Java currently does not have any task scheduling restrictions at task scheduling points.

3.1.6 Runtime Deployment

A deployment configuration is required upon launching the runtime. The deployment file specifies the number of workers, the number of places, and the mapping between workers and places and the mapping between places and hardware processors.

Figure 3.2 shows a sample locality-aware runtime deployment file. The first line specifies the number of workers and the number of places. For the 4 Quad-core Intel

```
16 8       // number_of_workers  number_of_places
0 @ 0 -> 0 4   // worker_id @ place_id --> processor ids...
1 @ 0 -> 0 4
2 @ 1 -> 8 12
3 @ 1 -> 8 12
4 @ 2 -> 1 5
5 @ 2 -> 1 5
6 @ 3 -> 9 13
7 @ 3 -> 9 13
8 @ 4 -> 2 6
9 @ 4 -> 2 6
10 @ 5 -> 10 14
11 @ 5 -> 10 14
12 @ 6 -> 3 7
13 @ 6 -> 3 7
14 @ 7 -> 11 15
15 @ 7 -> 11 15
```

Figure 3.2 : Locality-aware runtime deployment for a 4 Quad-Core Xeon SMP machine

Xeon SMP machine, the number of workers is set to 16 because there are 16 cores. Because each core pair shares a L2 cache, there are 8 separate L2 caches in the system. Thus, the number of places in the locality-aware deployment is chosen as 8. The rest of the file specifies the place of each worker and the processors that each worker is bound to.

3.2 Async-Finish Parallelism

Async-finish parallelism is defined as a class of computation that can be generated by async and finish constructs. The multithreaded computation model described in Section 2.2.1 is very general. In practice, we are interested in a particular class of computation dags that can be generated by a task-parallel language, and that can be

scheduled efficiently.

For async and finish constructs, as shown in Figure 3.1, we assume that task is started by a *spawn* instruction at an `async`, and instructions in the finish statement are enclosed by `startFinish` and `endFinish` instruction. The sync edge goes from the last instruction of a task to a `endFinish` instruction in one of its ancestors in the task spawn tree.

In HJ, a finish statement can only force the completion of descendant tasks [2]. As a result, all computations generated by HJ are strict computations. In HJ, however, tasks are synchronized at the `endFinish` instruction of its immediate enclosing finish instance, not at the end of its parent task. Thus the computation produced by HJ is not necessarily fully-strict. This is different from Cilk [15] and Cilk++ [9] in which all computations are fully-strict due to the implicit *sync* statement at the end of each cilk function.

In both Cilk and HJ, data dependencies are forced by task synchronization. While Cilk uses the `sync` keyword to synchronize all sub-tasks, HJ uses *finish* to synchronize all tasks spawned within the finish scope. Task synchronization forces the completion of the entire task, not a particular instruction. Thus the sync edge must start from the last instruction of a task. Therefore all computation dags generated by both Cilk and HJ are terminally-strict.

Although all computations generated by async-finish constructs are terminally-strict, the converse is not true. Figure 3.3 and 3.4 show computation dags that are terminally-strict, but cannot be generated by the async and finish constructs.

In Figure 3.3, Γ_3 spawns Γ_4 and Γ_2 synchronizes Γ_3. Because Γ_2 synchronizes Γ_3 through a join edge from v_{14} to v_{16}, then v_{16} must be `endFinish`. Since a finish instance synchronizes all sub-tasks that are created transitively, any sub-task of Γ_3 must be synchronized on or before v_{16}. If either v_5 or v_6 is `startFinish`, then Γ_4 should be synchronized to Γ_3. Otherwise, Γ_4 should be synchronized to Γ_2.

[2]The full Habanero-Java language also supports more general force operations for *futures* [42]

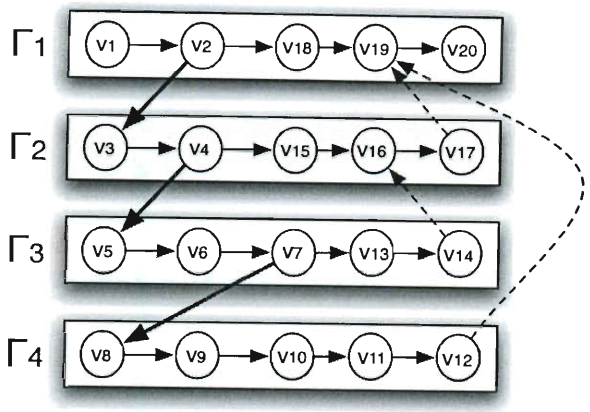


Figure 3.3 : This computation dag is terminally-strict computation but not async-finish computation

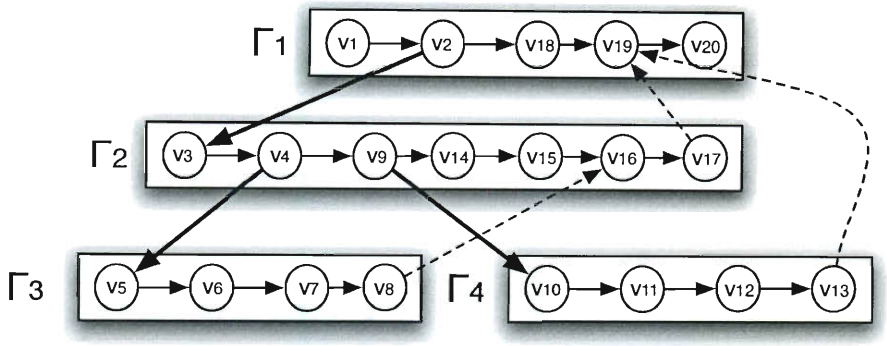


Figure 3.4 : This computation dag is terminally-strict computation but not async-finish computation

Therefore, it is impossible that Γ_4 is synchronized at Γ_1, and this computation cannot be generated by async-finish constructs.

In Figure 3.4, Γ_1 spawns Γ_2 and Γ_2 spawns Γ_3. Because Γ_3 is synchronized at Γ_2, v_3 must be `startFinish` and v_{16} must be `endFinish`. As a result, Γ_4 (spawned at v_9) must be synchronized to Γ_2, not Γ_1. Therefore this computation cannot be generated by async-finish constructs either.

In the rest of this section, we first present the definitions and notations for the *spawn tree* and *sync tree* for terminally-strict computation. We then summarize key properties of async-finish computations. These properties are used in the proof of the

space-bound of the adaptive scheduling algorithm in Section 5.2.

3.2.1 Spawn and Sync Trees

Given a computation dag, the *spawn tree* of the computation is defined as follows: the node set of its spawn tree contains all tasks, and task T_1 is task T_2's parent if there is a spawn edge from T_1 to T_2. Given a task γ, we use $ST_{spawn}(\gamma)$ to denote the set of tasks in γ's subtree (including γ) in the spawn tree of the dag and $PR_{spawn}(\gamma)$ to denote γ's spawn tree parent.

For terminally-strict computations , the *sync tree* can be defined as follows: each node in the sync tree corresponds to a dynamic instance of a task. There is an edge from task γ_a to γ_b in the sync tree if there is a *sync edge* in the dag from γ_b to γ_a. Given a task γ, we use $ST_{sync}(\gamma)$ to denote the set of tasks in γ's subtree (including γ) in the sync tree and use $PR_{sync}(\gamma)$ to denote γ's parent in the sync tree. The *sync tree* is well defined for terminally-strict computations because for each task except the root task, there is exactly one join edge goes from the last instruction to an `endFinish` instruction.

In fully-strict computations, the spawn tree is same as the sync tree. This is not the case for terminally-strict computations created by async-finish constructs. For example, Figures 3.6 and 3.7 show the spawn tree and the sync tree respectively for the HJ code listed in Figure 3.5. Notice that T_2 spawns T_3 at line 6, so T_2 is T_3's parent in the spawn tree. However, both T_2 and T_3 have the same sync tree parent T_1 because both T_2 and T_3 are spawned within the finish instance created at line 2 of task T_1, and they are synchronized at the end of finish instance at line 16 of T_1.

3.2.2 Properties of Async-Finish Computation

We summarize properties of async-finish computations. These properties are used in the proof of the space bound of the adaptive scheduling algorithm in Section 5.2.

Property 3.2.1. *If γ_a spawns γ_b, then either $PR_{sync}(\gamma_b) = \gamma_a$ or $PR_{sync}(\gamma_b) =$*

```
1     //T1
2   finish {
3         //T1
4       async {
5             //T2
6         async T3;
7         finish {
8               //T2
9           async T4
10        }
11      }
12      finish {
13            //T1
14          async T5
15    }
16  }
```

Figure 3.5 : HJ Code

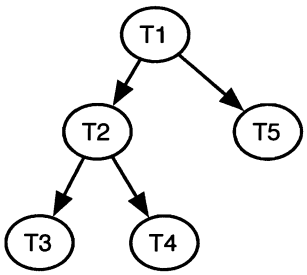


Figure 3.6 : Spawn Tree for the HJ Code Snippet in Figure 3.5

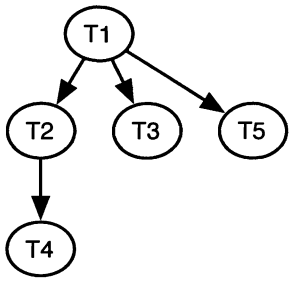


Figure 3.7 : Sync Tree for the HJ Code Snippet in Figure 3.5

$PR_{sync}(\gamma_a)$. *In both cases,* $\gamma_a \in ST_{sync}(PR_{sync}(\gamma_b))$.

Intuitively, the two cases in Property 3.2.1 corresponds to whether the parent task γ_a is in a finish scope or not.

Property 3.2.2. *If* $\gamma_b \in ST_{spawn}(\gamma_a)$ *and* $PR_{sync}(\gamma_b) = \gamma_a$, *for all* γ *such that* $\gamma_b \in ST_{spawn}(\gamma)$ *and* $\gamma \in ST_{spawn}(\gamma_a) - \{\gamma_a\}$, *we have* $PR_{sync}(\gamma) = \gamma_a$.

Property 3.2.2 follows directly from Property 3.2.1. It says if there is a spawn chain from γ_a to γ_b and γ_b syncs to γ_a, all tasks on the chain except γ_a should sync to γ_a.

The computation dag in Figure 3.3 is not an async-finish computation due to the violation of Property 3.2.1 and 3.2.2.

Property 3.2.3. *At any step in the schedule, when γ_a spawns γ_c, if there exists a live task γ_b such that $PR_{sync}(\gamma_b) = \gamma_a$, we have $PR_{sync}(\gamma_c) = \gamma_a$.*

The live task γ_b such that $PR_{sync}(\gamma_b) = \gamma_a$ implies that the dynamic IEF instance of γ_b is instantiated in γ_a and this instance will also synchronize all the child task of γ_b, e.g., γ_c.

The computation dag shown in Figure 3.4 is not an async-finish computation due to the violation of Property 3.2.3.

Chapter 4

HJ Work-Stealing Implementation

This chapter presents the implementation of the HJ work-stealing runtime called SLAW. The design is inspired by Cilk-style work-stealing [39], however, there are important differences in both compiler and runtime work-stealing support. The differences stem from the support for HJ language features, two task scheduling policies and locality-awareness. This chapter presents the implementation to support async-finish parallelism and two task scheduling policies. The locality-aware scheduling framework is presented in Chapter 6.

4.1 Work-stealing Task Scheduling Policies

In Blumofe's work stealing algorithm [15], when a task γ_a spawns task γ_b, the processor that spawns γ_b will start to work on γ_b eagerly, and the continuation of γ_a after the spawn might be stolen by other idle processors. This strategy is called the *work-first task scheduling policy.* As the alternative, the processor can stay on γ_a and let another processor help execute γ_b. the processor will work on γ_b later if γ_b is not picked up by other processors. This alternative strategy is called the *help-first task scheduling policy.*

The naming of the work-first and help-first can be understood from the perspective of the worker of the parent task. Under the work-first policy, the worker of the parent task will *work* on the child task *first.* Under the help-first policy, the worker of the parent task will ask other workers to *help* execute the child task *first.* The work-first policy is also called depth-first by some literatures because in 1-thread execution all the tasks will be executed in the order of depth-first traversal of the task spawn tree.

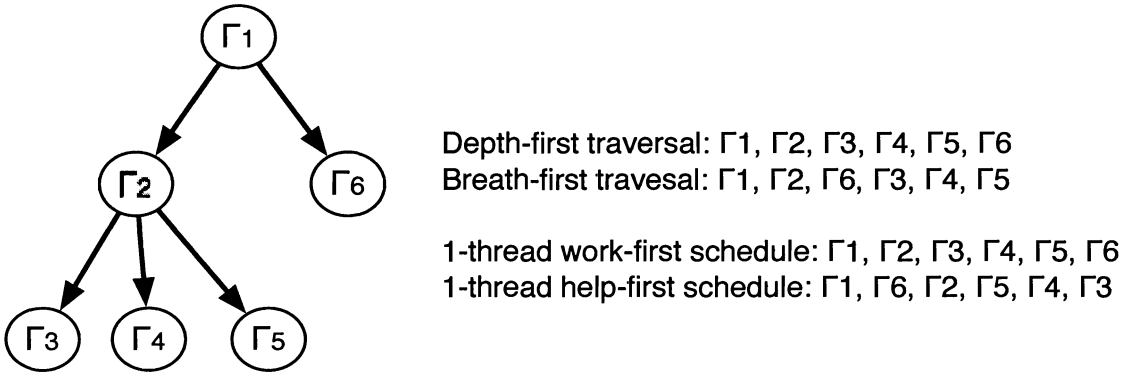


Figure 4.1 : Example of task scheduling under the work-first and the help-first policy vs. depth-first and breath-first traversal

Tasks scheduled under the help-first policy, on the other hand, are not executed in the breath-first order. Figure 4.1 shows a task spawn tree and the 1-thread execution order of the tasks under the work-first policy and the help-first policy.

This chapter presents the implementation of both task scheduling policies in a single work-stealing runtime. The study of task scheduling policies presented Chapter 5 discuss why we need to support both policies in a single runtime.

4.2 Escaping Asyncs and Finish Node

An *escaping async* is defined to be a task that may outlive its parent task. An escaping async can continue execution even after its parent task has terminated. Escaping asyncs are allowed in languages that produce terminally-strict computations but not fully-strict computations.

As an example, consider the Habanero-Java implementation of the parallel-DFS algorithm [23] shown in Figure 4.2. A single finish scope at line 19 suffices for all recursive descendant tasks spawned at line 14. It is possible for a call to `compute()` in a child task to outlive a call to `compute()` in a parent task. The only constraint is that all async calls to `compute()` must complete before the root task can continue execution past line 19. In contrast, if the algorithm is implemented in Cilk which generates fully-strict computations, the implicit `sync` operation inserted at the end

```
1  class V  {
2    V [] neighbors;
3    V parent;
4    V (int i) {super(i); }
5    boolean tryLabeling(V n) {
6       isolated if (parent == null)
7              parent = n;
8       return parent == n;
9    }
10   void compute() {
11      for (int i=0; i<neighbors.length; i++) {
12         V e = neighbors[i];
13         if (e.tryLabeling(this))
14            async e.compute(); //escaping async
15      }
16   }
17   void DFS() {
18      parent = this;
19      finish compute();
20   }}
```

Figure 4.2 : Code for parallel DFS spanning tree algorithm in HJ

of each task ensures that each parent task waits for all its child tasks to complete. In HJ terms, this would be equivalent to adding an additional finish scope that encloses the body of the `compute` function.

To support escaping asyncs in HJ, the *finish node* class is designed for task synchronization. Every worker maintains a finish node class instance representing the IEF of the next instruction to be executed. The compiler will transform each finish statement to a region of code bracketed by a pair of `startFinish` and `endFinish` statements. The `startFinish` statement is further transformed to a runtime API call to create a finish node. The `endFinish` statement is transformed to a runtime API call to perform the task synchronization of the finish node. Both `startFinish`

and `endFinish` change the finish node maintained by the worker.

Finish nodes are maintained in a tree-like structure with the parent pointer pointing to the node of its Immediately Enclosing Finish (IEF) scope. Apart from the parent pointer, each finish node has additional bookkeeping fields to ensure proper synchronization of tasks at the `endFinish`. When a worker is blocked at a `endFinish`, the continuation after the finish scope is saved in the finish node. This continuation is subsequently picked up for execution after tasks created within the finish scope are completed.

The details of the task synchronization protocol are described in Section 4.5.2.

4.3 Asynchronous Calling Convention

Besides escaping asyncs, HJ and Cilk differ in the asynchronous calling convention to parallel functions. From the language perspective, a function is a parallel if it may spawn a child task within its body or it calls other parallel functions. In Cilk, functions are classified as parallel *cilk functions* or sequential functions. Only a cilk function can be spawned as a child task and only cilk function may spawn tasks. Sequential calls to cilk functions are not permitted, though they can be simulated by spawning the function and then performing a sync operation immediately thereafter. This restriction has a significant software engineering impact because it increases the effort involved in converting sequential code to parallel code, and prohibits the insertion of sequential code wrappers for parallel code. In contrast, asynchronous tasks in HJ are created in an async statement, and the task can be any statement. HJ permits the same function to be invoked sequentially or via an async at different program points [1].

The program shown in Figure 4.3 is valid in HJ but cannot be directly translated to Cilk. In Cilk, `C()` and `E()` would be *cilk functions* because they (may) spawn tasks. Thus `C()` and `E()` cannot be called sequentially in function `B()` and `D()` respectively.

[1] Cilk++ [9] supports this capability as well

The reason for the strict asynchronous calling convection in Cilk is that it simplifies the implementation of saving and restoring the *execution context* of a task during in work-stealing. Similar to the execution context of a thread, the execution context of a task refers to all information need to be saved when a task's execution is interrupted and will resume later. With the calling convection restriction in Cilk, each invocation of a cilk function corresponds to a spawning of a task. Therefore, besides those global variables, only the PC and information in the activation record of the current function need to be saved in the execution context of the current task.

In HJ, the execution context of the current task needs to contain information in the activation record of the caller if the current function is called sequentially. This is because if stealing occurs in the parallel function, the thief will return from the callee and execute the statements after the call site. In other words, the continuation contains all information in the activation records of the current function and that of its serial callers in the call chain up to the function that is called asynchronously.

Consider the example in Figure 4.3. `C1` and `C2` label the points where continuations are pushed. At `C1`, the continuation pushed should contain the stack of activation records of function `C`, `B`, `A`. The thief that steals the continuation is responsible for starting the continuation at `C1` in function `C`. Upon returning from function `C`, the thief will resume the execution at `L2` of function `B`. The thief may again return to function `A` at `L1`. Similarly, the continuation pushed at `C2` should contain the activation records of function `E` and `D`.

A naive way to support a sequential call to a parallel function in a work-stealing runtime is to enclose the sequential call in `finish-async`. This approach reduces parallelism by disallowing the code after the sequential call to run in parallel with the task that escapes the callee of the sequential call. We are interested in solutions that do not entail unnecessary serialization of tasks.

One possible solution is to reconstruct the execution context lazily upon stealing. When the thief is stealing, it could first make a copy of the runtime stack of the

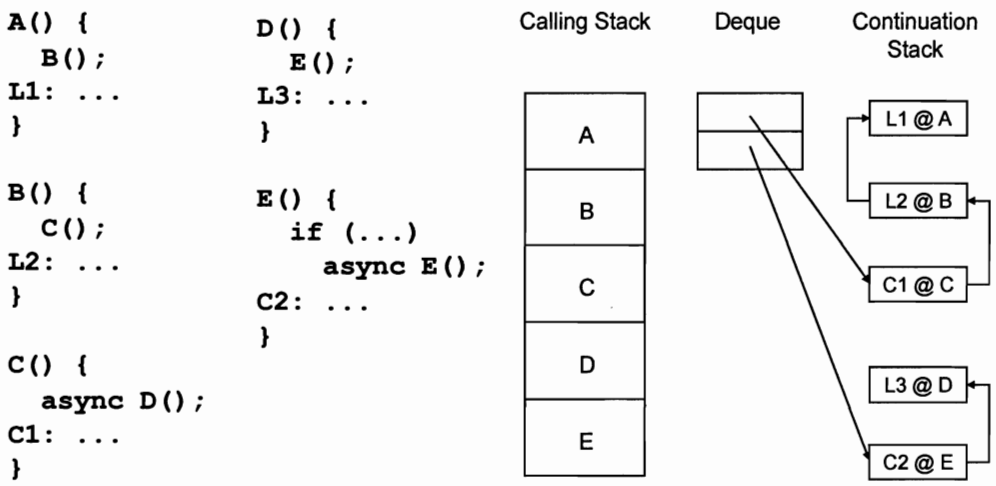


Figure 4.3 : Support for sequential call to a parallel function

victim before popping the frame out of the victim's deque. Then, the thief is able to reconstruct the continuation according to the copied stack before stealing and the content in the stolen frame. Ideally, this approach minimizes the cost of context saving, assuming the number of steals is few. However, this approach requires the thief to access the runtime stack of the victim. This approach is feasible in C-based systems but increases complexity in Java-based systems with managed runtimes. Further, this stack-copy this approach is not portable.

In HJ, we use a portable approach that is feasible for use in managed runtimes. The compiler will do whole program analysis to tag functions whose bodies, when invoked by a worker, may cause the worker to perform *context switch* before returning from the invocation. Context switches are defined formally in the context of computation dags in Section 5.1.1. Informally, from the runtime's persecutive, context switches are used to represent situations when the execution context of a worker is not handled automatically by the normal (sequential) calling convention at a task scheduling point. In this work-stealing implementation, context switches happen before the execution of every task spawned under the help-first policy or tasks stolen from other workers. Therefore functions need to be tagged are those contain non-trivial finish instances or spawn tasks under the work-first policy, as well as those

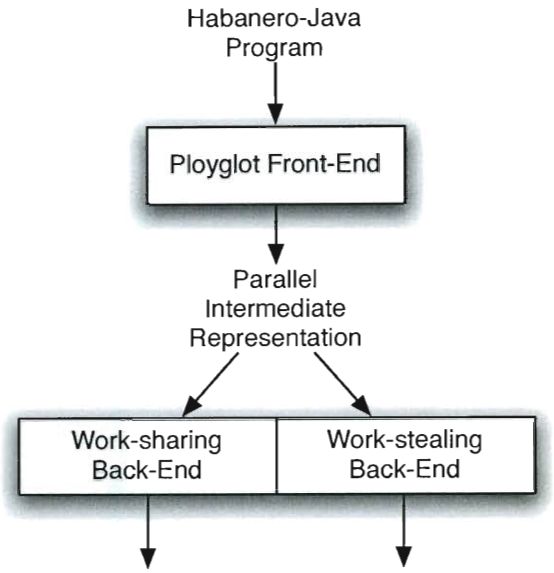


Figure 4.4 : HJ Compiler

that make sequential calls to tagged functions. Functions that only spawn tasks under the help-first policy without any synchronization is not tagged because the execution of those tasks is delayed after the return from the function invocation.

The compiler generates code to save the activation record of the current function before calling a tagged function. As shown in Figure 4.3, the continuation of a task at one point contains the stack of activation frames representing the activation records of functions in the sequential call chain, with the activation frame representing the activation record of the current function at the stack top. When the continuation is stolen, the thief will resume execution by calling a series of slow clones depending on the function represented by the activation frames in the continuation stack. The work-stealing compilation strategy and the compiler-produced code are illustrated in Section 4.4.

4.4 Work-stealing Compilation Strategy

Figure 4.4 shows the compilation process of the HJ program. HJ compiler is composed of a front-end and a back-end. The front-end uses polyglot to parse a HJ source code

into an abstract syntax tree (AST). The abstract syntax tree is then transformed to the parallel intermediate representation (PIR). The HJ compiler back-end produces code that runs on a specific runtime, which can be either work-sharing or work-stealing. This section describes the back-end work-stealing compilation strategy.

The HJ work-stealing compilation strategy is illustrated using the Fib example in Figure 4.5. This fib computes Fibonacci number recursively. In fib(n) at line 13, it spawns a child task fib(n-1) that can run in parallel with fib(n-2) in the parent task. The child task is synchronized before the sum of the return values can be retrieved at line 16. Because async statement does not return value, the return value is passed through a box integer as one of the parameters.

```
1  int Do(int n) {
2        BoxInt result = new BoxInt();
3        fib(n, result);
4        return result.value();
5  }
6  fib(int n, BoxInt result) {
7        if (n<2)
8           result.setValue(n);
9        else {
10           BoxInt x = new BoxInt();
11           BoxInt y = new BoxInt();
12           finish {
13              async fib(n-1, x);
14              fib(n-2, y);
15           }
16           result.setvalue(x.value()+y.value());
17 }
18 Main(int n) {
19        int result = Do(n);
20        System.out.println(result);
21 }
```

Figure 4.5 : Fib example in HJ using Integer Box to pass results

While the HJ compiler back-end produces bytecodes, we show their Java-equivalent for illustrative purpose in this thesis. Figures 4.7 to 4.10 show the Java-equivalent of the compiler generated code for the fib function with calls to the work-stealing runtime. Figures 4.11 and 4.12 show the compiler generated Java-equivalent code for the Do() and the main() function respectively.

For each function, the work-stealing compiler produces two classes: the function-specific activation frame class derived from the abstract ActivationFrame class (Figure 4.7) and the function-specific task wrapper class derived from the TaskWrapper class (Figure 4.8). For each function, the work-stealing compiler also produces two

```
1  public interface WorkerExecutable {
2    void execute(Worker worker) throws WorkerException;
3    FinishTreeNode getFinishScope();
4  }
5  public class ContinuationFrame implements WorkerExecutable {
6    public final ActivationFrame head;
7    private final FinishTreeNode _finishScope;
8    public ContinuationFrame(Closure c) {
9      this._finishScope = c.getFinishScope();
10     this.head = c.head;
11   }
12   public FinishTreeNode getFinishScope() {
13     return _finishScope;
14   }
15   @Override
16   public void execute(Worker worker) throws WorkerException {
17     ActivationFrame f = head;
18     worker.getClosure().head = f;
19     while (f != null) {
20       f.execute(worker);
21       f = f.next;
22     }
23   }
24 }
```

Figure 4.6 : Continuation Frame

static code versions: the fast clone (Figure 4.9) and the slow clone (Figure 4.10).

The function-specific activation frame class of function f contains the function parameters and local variables. The class represents the activation record of the function f on the runtime call stack. The function-specific activation frame implements the execute() function derived from the abstract super class. The execute() function takes the current worker as a parameter and calls the slow clone of the function f (line 37),

```
25 class runFibActivationFrame extends ActivationFrame {
26     // parameters
27     int n;
28     // Run function has a return value
29     Object retObject;
30     // local variables
31     BoxInt result;
32     public runFibActivationFrame(int n) {
33         this.n = n;
34     }
35     @Override
36      public void execute(Worker worker)  throws WorkerException {
37          runFibSlow(worker, this);
38      }
39 }
```

Figure 4.7 : Fib Activation Frame Class

```
40 class FibTaskWrapper extends TaskWrapper {
41     // parameters
42     public BoxInteger res;
43     public int n;
44     FibTaskWrapper(Worker worker, BoxInt res, int n) {
45        super(worker.getCurrentFinishScope());
46        this.res = res;
47        this.n = n;
48     }
49     @Override
50      public void execute(Worker worker) throws WorkerException {
51          Fib_Fast(worker, this.res, this.n);
52      }
53 }
```

Figure 4.8 : Fib Task Wrapper Class

```
54  private static void Fib_Fast(Worker worker, BoxInt res, int n)
55      throws WorkerException {
56    if (n < 2) {
57      res.v = n;
58      return;
59    }
60    boolean fast = true;
61    FibActivationFrame fibAf = new FibActivationFrame(res, n);
62    worker.beginMethod(fibAf);
63    worker.startFinish();
64    BoxInt x = new BoxInt();
65    if (use help first policy) {
66      worker.pushTaskWrapper(new FibTaskWrapper(worker, x, n - 1));
67      fast = false;
68    } else {
69      fibAf.x = x;
70      fibAf.pc = 1;
71      worker.pushContinuation();
72      Fib_Fast(worker, x, n - 1);
73      worker.popAndAbortOnSteal();
74    }
75                    ....
76    fibAf.x = x;
77    fibAf.y = y;
78    fibAf.pc = 3;
79    if (fast)
80      worker.endFinishFast();
81    else
82      worker.endFinishSlow();
83    res.v = x.v + y.v;
84    worker.endMethodFast();
85  }
```

Figure 4.9 : HJ Fib Fast Clone

```
86  private static void Fib_Slow(Worker worker, FibActivationFrame fibAf)
        throws WorkerException {
87    // restore local variables
88    BoxInt res = fibAf.res;
89    int n = fibAf.n;
90    BoxInt x = fibAf.x;
91    BoxInt y = fibAf.y;
92    switch (fibAf.pc) {
93    case 1:
94      y = new BoxInt();
95      if (use help first policy) {
96        worker.pushTaskWrapper(new FibTaskWrapper(worker, y, n - 2));
97      } else {
98        fibAf.x = x;
99        fibAf.y = y;
100       fibAf.pc = 2;
101       worker.pushContinuation();
102       Fib_Fast(worker, y, n - 2);
103       worker.popAndAbortOnSteal();
104     }
105   case 2:
106     fibAf.x = x;
107     fibAf.y = y;
108     fibAf.pc = 3;
109     worker.endFinishSlow();
110
111   case 3:
112     res.v = x.v + y.v;
113     worker.endMethodSlow();
114   }
115 }
```

Figure 4.10 : HJ Fib Slow Clone

and is invoked when a continuation containing f's activation frame is stolen.

Figure 4.6 shows the code of the Continuation class in the runtime. As described in Section 4.3, The continuation contains a list of ActivationFrames (line 6). The execute method unwinds and invokes the `execute()` method of every activation frame (line 17-22) in the continuation.

Since HJ compiler extracts (outlines) the async body into a separate method, the runtime assumes that the async body consists of a call to function f. The function-specific task wrapper class of function f represents function f as an asynchronous task. The execute function takes the current worker and original function parameters, and invokes the fast clone of function f (line 51). The execute function of a task wrapper is invoked when the task wrapper is stolen.

The fast clone of a function f runs when f is spawned under the work-first policy, called sequentially or called by f's task wrapper. When entering the fast clone of function f, an activation frame class function f is instantiated to represent the activation record of f on the runtime call stack (line 61). This activation frame instance is then saved in the execution context of the worker through the `beginMethod()` API call (line 62) and removed from the execution context through the `endMethodFast()` API call (line 84) when leaving the fast clone of function f. Each worker maintains a stack of activation frames to represent the current execution context, which is also called the closure. Upon spawning a child task, the specific action taken depends on the task scheduling policy: Under the help-first policy, a task wrapper of the asynchronous task function is instantiated and submitted to the runtime (line 66); Under the work-first policy, the current activation frame is saved (line 69,70), and the current closure of the worker is saved in a continuation frame that is pushed to the deque(line 71). After saving the current closure in the continuation frame, the closure is reset in `pushContinuation` method (line 71) before making a normal sequential function call to the asynchronous task function (line 72). When the call returns normally (not through an exception), the continuation frame is popped and

the closure is restored (line 73).

The slow clone is invoked when the continuation frame is stolen. The execute method of a continuation frame will pop the activation frames and invoke the execute method of the activation frames (line 17-22). In the beginning of the slow clone, local variables get their value from the activation frame (line 88-91). On exiting the slow clone, the `endMethodSlow` method is called to update the current closure of the worker. While return value is automatically handled through the calling convection in fast clones, a subtle issue arise when a value is returned in slow clones. When a function returns value in a slow clone, the return value is passed as a parameter to the `endMethodSlow` function. For example, because function Do returns an integer value, the `endMethodSlow` of the slow clone of Do function is invoked with the return value as the parameter at line 130. The `endMethodSlow` function will invoke the `setReturnResult` method of the caller's activation frame class - in this example - the `setReturnResult` of the `MainActivationFrame`. The `setReturnResult` method will set the proper field in the activation frame depending on the entry point.

4.5 Work-stealing Runtime Implementation

This section presents the implementation details of the HJ work-stealing scheduler. We will first discuss the implementation of the work-stealing deque, which is the core data structure for task stealing. Then we present the support for both work-first and help-first policies under a unified runtime.

4.5.1 HJ Work-stealing Deque

The HJ work-stealing deque implementation uses atomic compare-and-swap(cas) operations and is non-blocking. Compared to the algorithm described in Chase-Lev's dynamic circular work-stealing deque [21], this Java implementation will not cause memory leak.

While it is true that Java programmers do not need to deallocate objects, memory

leak is still a problem in Java. Java garbage collector collects unused objects automatically; however, the Java runtime thinks an object is unused only when it is dead, i.e., not reachable through reference from live objects.

The Java implementation of a simple non-concurrent queue shown in Figure 4.13 illustrates the Java memory leak problem. In this example, it is very important to set the queue cell to `null` at line 14. Otherwise, there may be a memory leak because the automatic garbage collector will not be able to collect the return object o, since the deque contains a reference to it.

Chase-Lev's dynamic circular work-stealing deque performs compare-and-swap operation on the deque top index pointer only without setting the corresponding deque entry to `null` (line 17, 32 in Figure 2.10 and 2.11). In work-stealing, the deque entry contains references to the activation frames, which in turn may contain references to large arrays. Failing to garbage collect this deque entries may cause a large portion of memory to be consumed by unused objects or arrays.

The HJ work-stealing deque implementation avoids this kind of memory leak by performing atomic compare-and-swap operations on the deque entry. The pseudo-code is shown in Figure 4.14 to 4.17. The worker successfully retrieves or steals the entry if the worker is able to atomically set the entry to `null`. The stealing conflicts are resolved by the compare-and-swap operation on the deque entry.

Resolving steal conflicts by performing cas operations on the entry requires that the same object is not pushed multiple times to the deque. This assumption holds for HJ work-stealing runtime. The objects pushed In HJ work-stealing, objects pushed to the deque are either continuations or task wrappers, each of which corresponds to a dynamic task spawn point. Either one continuation or task wrapper will be instantiated depending on the task scheduling policy used.

One subtle issue arises when supporting the dynamic growth of the circular array. As discussed earlier in Section 2.3.4, when the steal conflicts are resolved through the index pointer, the grow function can be simply implemented by copying the deque

entries from one deque to another, because the index of each entry remains the same in both deques. By resolving the steal conflicts through a cas operation on content of the deque entry, the grow function cannot simply copy the deque entry because it will create two deque entries with the same content, and two cas operations can both succeed. In the HJ deque implementation, the grow function will perform a cas operation on the old deque entry first and only copy the entry to the new deque if the cas returns true (line 13,14). This guarantees that the cas operation on the same entry in the old and new deque cannot both succeed. This increases the cost of the growth function. However, because the array size is doubled upon each growth, growth is expected to be a rare event.

4.5.2 Task Synchronization

As shown in Section 4.4, when a task is spawned under the work-first policy, the worker executes the child task and the continuation of the parent task after the spawn point is stored in the deque. When a task is spawned under the help-first policy, the worker stays on the parent task and a task wrapper of the spawned task is stored in the deque. Both continuations and task wrappers are considered *executable tasks*, or simply *executables* by HJ work-stealing runtime, and can be stolen by workers.

Each executable has a field of class type `FinishTreeNode` to represent its IEF. FinishTreeNode class, shown in Figure 4.18 is designed for task synchronization. All dynamic finish instances form a tree structure through the `parent` references. For each dynamic finish instance F, there is one *global active worker counter (*`gwc`*)*, and one local counter (`lc`) per worker. The global active worker counter counts the total number of workers that are currently executing executables whose IEF=F. The local counter for a worker counts the number of executables that are stored but yet to be executed. All counters are volatile and atomic. The global active worker counter has a version number (`_gwcVersion`) which advances when the `gwc` is incremented from 0 to 1. The version number is used to detect the ABA problem of the global active

worker counter as discussed later in this Section. When a task waiting for child tasks is suspended at `endFinish` of a finish instance, the continuation after `endFinish` is stored in the finish instance.

We use two-level (global and local) counters in the task synchronization protocol: the global counter counter only counts active workers, not tasks, and the local counter for each worker counts tasks pending execution. Compared to having a single global counter that counts all incomplete tasks, this design reduces synchronization overhead on the counters. According to the task synchronization protocol described later, both the global and local counters may be updated by multiple workers. The synchronization overhead on the atomic counters depends on the level of conflicts, which is directly related to the number of steals. In fact, the updates of the global active worker counter is triggered by steals and the local counter of a worker is only accessed by the worker itself and the thief upon stealing. As discussed before in Section 2.3.3, for programs with abundant parallelism, the number of steals is small compared to the number of tasks. As a result, the synchronization overhead is small for programs with abundant parallelism.

Figure 4.19 and 4.20 lists the pseudo-code of the task synchronization protocol. After initialization, one worker start executing the main function, which begins with a `startFinish` operation. Other workers start stealing. The runtime terminates when the continuation after the main finish scope is executed.

We define the term *check-in* and *check-out* for a worker and a finish instance. A worker *checks into* a finish instance F if it enters F by calling `startFinish` or it begins to execute a executable of F after stealing it from the other worker. A worker *checks out of* a finish instance F when (1) it completes a executable under F or it is blocked at `endFinish`, and (2) there is no local executable under F on the local deque. If there is still a local executable under F, the worker will defer the check-out until all local executables under F are completed. Note that when a worker checks into a new finish instance F by calling `startFinish` of F, it will not check-out of F's

parent, but instead, will choose the worker that executes the continuation after F as *its delegate* to check-out F's parent. The check-in and check-out operation increments (or initialize) and decrements the global active worker counter respectively. For each check-in operation of F, there is exactly one check-out operation of F, though they may be performed by two different workers through delegation.

Now we argue that the runtime guarantees that for any finish scope F, the continuation after F is *safely* executed by *exactly one* worker:

- First, we argue that when `verifyComplete(F)` returns true at line 33, it is *safe* to execute the continuation after F, i.e., all tasks spawned within the finish scope F have been completed. When `verifyComplete` returns true, it verifies that all workers checked into the finish scope have been checked out and there is no worker that is holding a stolen task but has not checked in yet. The former case is detected by verifying that the global worker counter is 0 as there is exactly one check-out for each check-in. The latter is detected by comparing the version number of the global worker counter before and after verifying all local task counters are 0. Note that in the steal function, the thief checks in (line 4) *before* the local task counter of the victim is decremented (line 5). If there is a worker that steals a task but has not yet checked in, the local task counter of the worker must be greater than 0. Observe that when a worker checks out of F, there is no task under the F on its local deque. So when the global counter of F is 0 and no stealing is happening, it is safe to execute the continuation after F.

- Second, we observe that `verifyComplete` will return true after the last worker decrements the global worker counter to 0. The `CAS` operation ensures that at most one worker will execute the continuation in case there are multiple workers competing for the continuation.

4.5.3 Optimization

Optimizations are inspired by the work-first principle, i.e., to reduce the overhead of singled-threaded execution assuming there is no stealing.

One optimization is to lazily create the array of atomic local counters. Each FinishTreeNode has a local counter for the worker that calls `startFinish`. The whole array of local counters are created only upon the first steal. In 1-thread execution, since there is no stealing, the array is not allocated at all.

Another optimization is to split the atomic local counters per worker into a private local counter and a public steal counter. The private local counter is only accessed by the worker locally. The public steal counter of the worker is incremented when each time the worker becomes a victim of a steal. The testing of local counter with 0 in the previous protocol (line 42) can be substituted with checking if public steal counter equals private local counter. In 1-thread execution, since there is no stealing, only the private local counter will be incremented and decremented. The private local counter does not need to be atomic thus does not trigger a lock on the bus.

```
116   private static int Do_Fast(Worker worker, int n) throws
          WorkerException {
117     DoActivationFrame doCf = new DoActivationFrame(n);
118     worker.beginMethod(doCf);
119     BoxInt ret = new BoxInt();
120     doCf.ret = ret;
121     doCf.pc = 1;
122     Fib_Fast(worker, ret, n);
123     worker.endMethodFast();
124     return ret.v;
125   }
126   private static void Do_Slow(Worker worker, DoActivationFrame doCf)
          throws WorkerException {
127     BoxInt ret = doCf.ret;
128     switch (doCf.pc) {
129     case 1:
130       worker.endMethodSlow(ret.value());
131     }
132   }
133   public final static class DoActivationFrame extends ActivationFrame {
134     int n;
135     BoxInt ret;
136     public DoActivationFrame(int n) {
137       this.n = n;
138     }
139     public void execute(Worker worker) throws WorkerException {
140       Do_Slow(worker, this);
141     }
142   }
```

Figure 4.11 : Compiler generated code for Do function

```
143  public final static class MainActivationFrame extends ActivationFrame
         {
144    int ans;
145    int n;
146    public MainActivationFrame(int n) {
147      this.n = n;
148    }
149    public void execute(Worker worker) throws WorkerException {
150      main_Slow(worker, this);
151    }
152    public void setReturnResult(Object v) {
153      switch (this.pc) {
154      case 1:
155        this.ans = (Integer) v;
156        return;
157      }   } }
158  private static void mainSlow(Worker worker, MainActivationFrame mainAf
       ) throws WorkerException {
159    int n = mainAf.n;
160    int ans = mainAf.ans;
161    switch (mainAf.pc) {
162    case 0:
163      mainAf.pc = 1;
164      ans = Do_Fast(worker, n);
165    case 1:
166      System.out.println(ans);
167      worker.endMethodSlow();
168    }
169  }
```

Figure 4.12 : Compiler generated code for main function

```
1   class Queue {
2      Object [] array;
3      int size;
4      int head, tail;
5      ...
6      public void enqueue(Object o) {
7          array[tail] = o;
8          tail = (tail +1) % size;
9      }
10    public Object dequeue() {
11         if (isEmpty())
12            return null;
13         Object o = array[head];
14         array[head] = null;  // there may be memory leak without this
               line
15         head=(head+1)  % size;
16         return o;
17     }
18  }
```

Figure 4.13 : Java queue implementation

```
1  public class HJWstDeque {
2    class CircularArray {
3      private int size;
4      private AtomicReferenceArray<Object> segment;
5      boolean casEntry(int i, Object expect, Object update) {
6        int index = i % size;
7        return this.segment.compareAndSet(index, expect, update);
8      }
9      CircularArray grow(int b, int t) {
10       CircularArray a = new CircularArray(this.log_size + 1);
11       for (int i = t; i < b; i++) {
12         final Object o = this.get(i);
13         if (this.compareAndSet(i, o, null)) {
14           a.put(i, o);
15         } else
16           a.put(i, null);
17       }
18       return a;
19     }
20     }
21   }
22   private volatile int bottom = 0;
23   private volatile int top = 0;
24   ....
25 }
```

Figure 4.14 : HJ Work-stealing Deque class, Circular Array class and the grow function

```
26 public void pushBottom(Object o) {
27  long b = this.bottom;
28  long t = this.top;
29  CircularArray a = this.activeArray;
30  long size = b - t;
31  if (size >= a.size()-1) {
32  a = a.grow(b, t);
33  this.activeArray = a;
34 }
35  a.put(b, o);
36  bottom = b+1;
37 }
```

Figure 4.15 : HJ work-stealing deque pushBottom

```
38 public Object popBottom() {
39     int b = this.bottom;
40     CircularArray a = this.activeArray;
41     b = b - 1;
42     this.bottom = b;
43     int t = this.top;
44     int size = b - t;
45     if (size < 0) {
46       bottom = t;
47       return Empty;
48     }
49     Object o = a.get(b);
50     if (size > 0) {
51       a.put(b, null);
52       return o;
53     }
54     if (o == null || ! a.casEntry(t, o, null)) {
55       this.bottom = t + 1;
56       return Empty;
57     }
58     return o;
59   }
```

Figure 4.16 : HJ work-stealing deque popBottom

```
60   public Object steal() {
61     int t = this.top;
62     int b = this.bottom;
63     CircularArray a = this.activeArray;
64     int size = b - t;
65     if (size <= 0)
66       return Empty;
67     Object o = a.get(t);
68     if (o == null)
69       return Abort;
70     if (!a.casEntry(t, o, null)) // here we assume frame is not re-used.
71       return Abort;
72     top = t+1;
73     return o;
74   }
```

Figure 4.17 : HJ work-stealing deque steal

```
1 class FinishTreeNode {
2   FinishTreeNode parent; // reference to dynamic parent finish instance
3   AtomicInteger gwc; // global active worker counter
4   int _gwcVersion; // version number of gwc
5   ContinuationFrame suspendedContinuation; // suspended continuation
6   AtomicInteger[] lc; // local executable counters
7   ...
8 }
```

Figure 4.18 : FinishTreeNode class

```
1    function Object steal () {
2        task = steal task from victim's deque;
3        finish = task's finish scope;
4        current worker checks in under finish;
5        finish.lc[victim]--;
6        return task;
7    }
8    function push_task_to_deque(task) {
9        finish = current finish scope;
10       finish.lc[this_worker]++;
11      this.deque.pushBottom(task);
12   }
13    function check_in(finish) {
14       if (finish.gwc.getAndIncrement() == 0);
15           finish.gwc.version++;
16   }
17   function check_out(finish) {
18       decrement finish.gwc;
19   }
20   function startFinish() {
21       checks in new finish scope;
22   }
23   function endFinish() {
24       finish = current finish scope;
25       save continuation after finish;
26       return to runtime;
27   }
```

Figure 4.19 : Task Synchronization Protocol

```
28   function task OnTaskComplete() {
29      finish = current finish scope;
30      task = get_local_task under finish;
31      if (task != null) return task;
32      check_out finish;
33      if (verifyComplete(finish)) {
34          if (CAS(finish.gwc, 0, -1)) {
35            return finish.continuation;
36      }
37      return get_local_task;
38   }
39   function boolean verifyComplete(finish) {
40       versionOld = finish.gwc.version();
41       if (finish.gwc != 0) return false;
42       if (not all lc of finish 0)
43          return false;
44       versionNew = finish.gwc.version();
45       return versionOld == versionNew;
46       }
```

Figure 4.20 : Task Synchronization Protocol (cont)

Chapter 5

Adaptive Work-stealing

This chapter studies the pros and cons of both work-first and help-first task scheduling policies. The conclusion is that both policies have pros and cons that are complimentary to each other in different scenarios. This study motivates the design of the adaptive scheduling algorithm used in HJ work-stealing runtime. This chapter is concluded by experimental results of HJ work-stealing runtime.

5.1 Study of Task Scheduling Policies

This section presents the study of the work-first and help-first policies under different scenarios: recursive parallelism and flat parallelism. Two aspects considered are the performance and the memory usage. For performance, we consider the context switch as the major source of overhead and analyze the cost of context switches in both policies. For memory usage, we focus on the stack usage and heap usage.

5.1.1 Context Switch

We define context switch under the task scheduling model described in Section 2.2.1. Given a schedule for a multithreaded computation dag, a context switch is said to occur between two consecutive instructions a and b, if

1. a and b is not connected by a continue edge, AND
2. b is the instruction that will be executed after a in the serial depth-first schedule, when spawn edge is considered a sequential call the child task.

For example, for the computation dag shown in Figure 2.1, the instructions are numbered in the order of serial depth-first execution. The schedule $...v_{12} -> v_{13} -> v_{14} -> v_{18}...$ has a context switch between v_{14} and v_{18} because v_{14} and v_{18} is not connected by a continue edge and, under serial depth-first execution, v_{15} not v_{18} should be executed after v_{14}. In another schedule $v_1 -> v_2 -> v_3 -> v_6 -> v_9 -> v_4 -> v_5$, there is a context switch between v_9 and v_4.

Context switches are a major source of overhead in a work-stealing scheduler [82] for the following reasons. First, when there is a context switch, the execution context of a worker is not handled automatically by the normal (sequential) calling convention at a task scheduling point. In HJ, the call stack is cleared during the context switch by throwing an exception that is to be caught by the scheduler. Second, it is believed that there is inherent data locality in serial depth-first execution and deviations may incur cold cache misses [3].

5.1.2 Recursive and Flat Parallelism

Recursive parallelism and flat parallelism are two common parallel patterns in task parallelism. In recursive parallelism, problems are recursively decomposed and solved using the divide-and-conquer approach. From the computation dag's perspective, the number of total number of tasks grows exponentially to the depth of the spawn tree. Therefore, *parallelism (T_1/T_∞) is typically abundant in recursive parallelism.* For example, in the two-way recursive `fib(n)` example in Figure 2.12, the total number of tasks created is $O(2^n)$, of which only n tasks are on the critical path. the parallelism is $O(2^n/n)$.

In flat parallelism, on the other hand, tasks are created iteratively. From the computation dag's perspective, the depth of the spawn tree is small as most tasks spawned are leaf tasks. For a program that iteratively spawn N tasks (as shown in Figure 5.1), the total number of task is N and the length of critical path is one task (ignoring the scheduling overhead). Assuming each task contains equal amount of

work, the parallelism of this program is $O(N)$. Compared to recursive parallelism in which parallelism is typically abundant, flat parallelism may not have abundant parallelism.

Recursion is a powerful and elegant way to decompose problems and express parallelism. Many HJ programs are written in the recursive style. Iterative parallelism, on the other hand, is common in pointer-chasing programs and loop parallelism. In practice, many graph or tree algorithms are a combination of both. For example, in the parallel depth-first search algorithm shown in Figure 4.2, the depth-first search algorithm is recursive, but for each node, the child tasks are spawned iteratively. Depending on the shape of the input graph and the order of visit, the task spawn tree is irregular. In order to perform well in those kind of algorithms, it is desirable to tune the runtime performance on both recursive and flat parallelism.

5.1.3 Performance

Under the work-first policy, if there is only 1 worker thread, the worker will execute all tasks in the same order as the equivalent sequential program. This results in no context switch and reduced 1-thread execution time. Since the work-first principle focuses on optimizing the 1-thread execution, the Cilk runtime whose design is prevailed by the work-first principle, uses the work-first policy for all spawned tasks. If there are multiple workers, because every stolen tasks are executed after a context switch, the number of context switches equals the number of steals.

Under the help-first policy, because the worker executes the tasks lazily only at the task synchronization point, every tasks are executed after a context switch.

We now consider the performance of both policies with respect to the number of steals and total number of tasks. When the number of steals is relatively small compared to the total number tasks, the work-first policy performs well in performance due to few context switches.

We consider the situation when the number of steal is relatively high. Consider a

```
1 // T1
2 async S2;
3 async S3;
4 ...
5 async SN;
6 S1;
```

Figure 5.1 : Task T1 spawns N-1 tasks consecutively.

scenario in which one busy worker creates $N-1$ tasks consecutively (code shown in Figure 5.1) and the other $N-1$ workers are idle and polling for tasks. This scenario leads to high stealing rate because any frame pushed to the deque is expected to be stolen immediately. Let the total amount of work in each task is T, the cost of push operation in t_{push} and the amount of time to migrate a task from one worker to another be $t_{migrate}$. $t_{migrate}$ consists of two parts: the time spent in the steal function of the the deque t_{steal} and the time to do context switch before executing the new task t_{cs}. Under the work-first policy, all $N-1$ task migrations are serialized. The total amount of work under the work-first policy:

$T_1^{wf} = (t_{push} + t_{steal} + t_{cs}) * (N-1) + NT$

and the length of the critical path:

$T_\infty^{wf} = (t_{push} + t_{steal} + t_{cs}) * (N-1) + T.$

Under the help-first policy, although the tasks have to be popped from the top-end of the victim's queue in order, the context switches can be done in parallel. In the scenario described above, assuming $t_{push} < t_{steal}$ (which is usually true since steal is more expensive due to steal conflicts), the critical path of a help-first execution:

$T_\infty^{hf} = t_{push} + t_{steal} * (N-1) + t_{cs} + T.$

The total amount of work under the help-first policy as the same as the work-first policy:

$T_1^{hf} = T_1^{wf} = (t_{push} + t_{steal} + t_{cs}) * (N-1) + NT.$

This analysis suggests that the work-first policy tends to prolong the critical path

by serializing context switches (t_{cs}). The prolonged critical path may not affect the performance for recursive parallelism where parallelism is abundant and the low stealing rate is low. But for flat parallelism where parallelism may already be low, the prolonged critical path could significantly degrade the performance. This analysis is confirmed by the experimental result shown in Section 5.3.

5.1.4 Memory Issue

Performance is not the only concern when building a task scheduler. Another important consideration is the resource bound. The implementation of the work-first and the help-first policy can result in different stack and heap bound.

It is well known that work-stealing schedulers that use the work-first policy, such as the Cilk work-stealing scheduler, are provably space-efficient [15]. If the serial depth-first execution of parallel application uses S_1 memory, the memory usage of the P-processor execution is bounded by S_1P. However, the implementation of the work-first policy results in using more stack space than necessary. According to the work-first principle, in order to minimize the 1-thread execution time, the work-first policy is almost always implemented as a sequential call to the child task. This implementation executes the child task without releasing the existing execution context although the child and parent task can run in parallel. Therefore, the stack usage is the same as the depth-first serial execution.

The stack problem becomes a significant disadvantage of the work-first policy for many recursive algorithms. For example, in the Parallel Depth First Search (PDFS) benchmark shown in Figure 4.2, *the depth of the task spawn tree is proportional to the program's data size.* Work-stealing schedulers using the work-first policy will terminate prematurely due to stack overflow for large graphs because of the stack limit in current environment. Although it is possible to instead allocate stack frames in the heap, these solutions are mostly OS and architecture dependent, and cannot be used in a managed runtime such as our portable Java-based implementation.

On the other hand, the help-first policy can be used to reduce the maximum stack size constraint for workers. Tasks spawned under the help-first policy are executed after context switches. In HJ work-stealing runtime, an exception is throw and caught by the runtime scheduler during the context switch. Such a context switch has the effect of releasing the stack pressure. In fact, if all activation frames are assumed to be constant in size, executing computation dag under the help-first policy in HJ only requires constant stack size per worker.

Despite their stack bound, work-scheduling schedulers that use the help-first policy are not provably space-efficient. In the worst case, if the parent task spawns an unbounded number of child tasks, all these child tasks will be saved to a deque on the heap, which may lead to heap overflow. However, the serial depth-first execution of the same program runs with bounded memory.

Stack pressure is a concern for recursive programs even in serial depth-first execution. One of the assumption of Cilk is that the sequential execution completes successfully. Under this assumption, some graph algorithms, e.g. the parallel depth-first execution, cannot be written in the recursive style because the sequential version will also overflow the stack. This limits the expressiveness of the language because divide-and-conquer is a powerful and elegant approach to decompose the problem and divide-and-conquer is usually written in recursive style.

5.2 Adaptive Scheduling Policies

The study of both performance and memory issue suggest that the work-first and the help-first policy have pros and cons that are complimentary to each other in different scenarios. This study motivates the design of the adaptive scheduling algorithms used in SLAW, the work-stealing runtime for HJ. There are two major scalability concerns when designing the adaptive scheduler: (1) establishing space bounds which include stack space for worker threads as well as the total memory space; and (2) selection of help-first and work-first policy for better performance in different scenarios.

As described in Section 5.1.4, the work-first policy increases the stack pressure and tasks spawned under the help-first policy are executed after releasing the stack pressure via context switching. Let us assume that S is the space limit (or *threshold*) for a worker's stack. If the input program has a spawn tree depth greater than S, then it is necessary at some point to use the help-first policy to ensure that a worker's stack space does not exceed threshold S. This decision is presented as *stack condition* in the spawn rule for Algorithm 1 discussed later.

Besides the stack bound, we also consider the total memory bound. The total memory bound is determined by the memory usage of both started and fresh tasks. Started tasks are those that have been executed by some processor; fresh tasks have been spawned but never executed. When only spawning under work-first policy, there will be no fresh tasks and the total memory bound of started tasks has been established by past research on work-stealing schedulers [15, 5]. However, under the help-first policy, all child tasks will be created as fresh tasks and saved on the heap. In order to provide a total memory guarantee for the adaptive work-stealing scheduler: the scheduler must switch to the work-first policy when the number of fresh tasks exceeds a threshold; this ensures that the total memory used by fresh tasks are bounded. The threshold is called the *fresh task threshold* denoted as F. This decision is presented as *fresh task condition* in the spawn rule for Algorithm 1.

These two conditions are enough to establish the stack and total memory bounds for the adaptive scheduling algorithm. One thing that is important to notice is that the adaptive scheduler treats stack bound as a hard bound and gives the stack condition higher priority than the fresh task condition. When the stack threshold is reached, help-first policy will always be used to avoid stack overflow regardless of the number of fresh tasks created.

SLAW employs a runtime heuristic to select the policy if neither of these two conditions is met. This heuristic is not required to establish the worst-case stack and memory space bound, but is designed to achieve better scalability and performance

in practice. For this reason, the heuristic is described below but is not presented in the algorithm.

Before describing the heuristic, we first discuss two techniques used to reduce the overhead of adaptation and evaluation of the task spawning policy. First, each worker maintains its own spawning policy and the heuristic used to evaluate the spawning policy consists only of thread-local operations. We show in the Section 5.3.2 that the overhead is lower than 5%. Second, SLAW does not re-evaluate the spawning policy at every spawning point. Instead, it starts with the help-first policy at the beginning and re-evaluate the spawning policy periodically at an interval for every INT spawned tasks. The reason that it starts with the help-first policy is because steals are usually frequent at the beginning of the application, and the help-first policy performs better than the work-first policy when stealing is frequent. Evaluating the policy periodically further amortized the overhead.

The heuristic used by SLAW is based on a simple estimation on the likelihood of the new spawned task being stolen. It computes the number of tasks that were stolen from the worker during the last interval. If the number of steals is greater than INT, this implies the steal rate is higher than the task creation rate. The scheduler will use the help-first policy for the new task in the next interval to increase the rate of distributing tasks to other workers. Otherwise, the scheduler assumes the new task will not be stolen and thus uses work-first policy for the next interval to reduce the overhead of context switches.

In summary, the thresholds S and F for the *stack condition* and the *fresh task* condition are used to bound the algorithm's stack and total space requirements respectively. The third parameter INT is used to control the policy re-evaluation interval in SLAW to reduce overhead. Later in Section 5.3.2, we present experimental results on the sensitivity of performance to these parameters and also discuss selection of their default values.

5.2.1 Taxonomy

We use the notation $G(V, E, S_1)$ to represent a computation *dag* with set of tasks V tasks, set of edges E, and stack depth S_1 which is defined as the total memory space required for serial depth-first execution.

A P-processor schedule X of a dag is defined as a sequence of steps, where each step consists of at most P instruction, one for each processor. For a given dag, $X(t, p)$ denotes the task of the instruction executed by processor p at step t.

Task terminates: a task γ terminates after step t, if its last instruction is executed at step t by processor p. We also say processor p finishes/terminates γ at step t.

Suspended task: A task γ is suspended before step t if γ cannot be run at step t due to dependency. We say processor p suspends a task γ at step t if $\gamma = X(t, p)$ and γ is suspended before step t+1.

Task ready: A task is ready if it is not suspended. A task *becomes* ready before step t if it is suspended before step t-1 is ready before step t.

Fresh task: A task γ is said to be *fresh* before step t if it has been spawned by processor p before step t, but is never executed by any processor. Processor p is called γ's owner. Fresh tasks are saved on the heap. We assume the size of all fresh tasks is $O(1)$.

Preempted task: Task γ is *preempted* by processor p *at* step t if $\gamma = X(t-1, p)$ and γ is not suspended before step t and $\gamma \neq X(t, p)$. No task is preempted at step 0.

Task γ is *preempted and owned* by processor p before step t if:

1. $\gamma \neq X(t-1, q)$ for any processor q

2. γ is either preempted and owned by p before step t-1 or preempted by processor p at step t-1 .

In the adaptive algorithm, a task is preempted only when it performs a work-first spawn (Action 1 in Algorithm 1).

Making progress: A task γ is making progress due to processor p at step t if $X(t,p) \in ST_{spawn}(\gamma)$. A task is said to be making progress at step t if it is making progress due to any processor at step t.

Progressive schedule: We say a schedule is progressive if all non-fresh tasks are making progress for every step.

Theorem 5.2.1. *For any P-processor progressive schedule X for a dag $G(V, E, S_1)$, for any step, the memory usage of all non-fresh tasks is bounded by S_1P.*

Proof. There are at most P leaf tasks in the spawn sub tree composed by all non-fresh tasks at any step. Otherwise, at least one task does not make progress. The space used by the activation frames of a leaf task and all its ancestors are bounded by S_1. So the total memory usage of all non-fresh tasks is bounded by S_1P. □ □

Theorem 5.2.1 provides the bound for non-fresh tasks in a progressive schedule. If the memory usage of fresh tasks is also bounded, the total memory space will be bounded. In the following subsection, we present our adaptive scheduling algorithm which is progressive and has a bound for non-fresh tasks if the stack threshold is not exceeded.

5.2.2 Scheduling Algorithm

We use P-ADP(S, F) to denote a P-processor adaptive schedule that can be generated by the adaptive work-stealing algorithm shown in Algorithm 1. As mentioned earlier, S and F denote the stack threshold and fresh-task threshold respectively.

To abstract the runtime call stack, some tasks are flagged *on-stack-p* where p is a processor id. The activation records of those tasks flagged on-stack-p are considered to be on processor p's runtime call stack. The algorithm always flags a task γ as *on-stack-p* if processor p starts executing γ. This flag will not be cleared when γ spawns a new task under the work-first policy, since work-first task spawn is implemented as a sequential call in SLAW. However, when a processor p does a context switch to start executing a fresh-task or a suspended task, all *on-stack-p* flags for that particular processor p are cleared.

Actions 1-5 in the algorithm and the idle routine, guarantee that all adaptive schedule executes tasks in a depth-first order when a task is suspended or terminates. However, when many tasks have the same depth, tie breakers in the idle routine are important to ensure progressive-ness of the schedule. This leads the space bound of the algorithm. The stealing restriction is also important to establish the space bound.

We present the following lemmas for the adaptive algorithm. These two lemmas are used to prove the progressive-ness of the adaptive schedule.

Lemma 5.2.2. *Given any adaptive schedule X, for any processor p and step t, if a task γ is preempted and owned by p before step t, then either $X(t,p) \in ST_{spawn}(\gamma)$ or $\gamma = X(t,q)$ for some processor q.*

Lemma 5.2.3. *Given any adaptive schedule X, for any processor p and step t, if fresh task γ is created and owned by processor p before t, then either $X(t,p) \in ST_{sync}(PR_{sync}(\gamma))$ or $\gamma = X(t,q)$ for some processor q.*

Proof. The proof uses the properties of async-finish computation described in Section 3.2.2.

We prove Lemma 5.2.2 and 5.2.3 together by induction on the time step by enumerating situations when Actions 1-5 or the idle routine are taken by the algorithm for each step. Both lemma arguments can be verified true for step 0 and 1. Assume both arguments are true for all tasks and processors at all steps at or before t, we

show both arguments are true for any preempted task γ_{pt} owned by processor p and any fresh task γ_f created by processor p at step t+1.

Here are some denotations used throughout the proof: $\gamma_t = X(t,p)$, $\gamma_{t+1} = X(t+1,p)$, $\gamma_z = PR_{spawn}(\gamma_f)$, $\gamma_y = PR_{sync}(\gamma_f)$, $\gamma_q = PR_{spawn}(\gamma_{t+1})$ and $\gamma_s = PR_{sync}(\gamma_{t+1})$. Note according to Property 3.2.1, for γ_q and γ_s, either $\gamma_q = \gamma_s$ or $PR_{sync}(\gamma_q) = \gamma_s$. Similarly for γ_z and γ_y, either $\gamma_z = \gamma_y$ or $PR_{sync}(\gamma_z) = \gamma_y$.

To show the argument for Lemma 5.2.2 is true for step t+1, we need to show $\gamma_{t+1} \in ST_{spawn}(\gamma_{pt})$ or $\gamma_{pt} = X(t+1,q)$ for some processor q. To show the argument for Lemma 5.2.3 is true for step t+1, we need to show $\gamma_{t+1} \in ST_{sync}(\gamma_y)$ or $\gamma_f = X(t+1,q)$ for some processor q. In following proof, we assume both γ_f and γ_{pt} are not stolen. If they are stolen, they will be executed at step t+1 by some processor q. We show both arguments are true separately.

We first show the argument for Lemma 5.2.2 is true. Because γ_{pt} is preempted and owned by p before step t+1, by the definition of *preempted and owned*, γ_{pt} must be either preempted and owned by p *before* step t or preempted by p *at* step t.

If γ_{pt} is preempted by p at step t, p must perform a work-first spawn (Action 1) at step t. Thus $\gamma_{pt} = PR_{spawn}(\gamma_t)$. If γ_{pt} is preempted by p before step t, by the assumption, γ_{pt} must be making progress due to p at step t (γ_{pt} is not suspended before step t and no processor can execute γ_{pt} at step t because γ_{pt} is suspended and owned by p before step t+1). In both cases, we have $\gamma_t \in ST_{spawn}(\gamma_{pt})$.

Now we enumerate the situation when Actions 1-5 are taken by the Algorithm 1 at step t+1 and the situation when the idle routine is called before step t+1 to remove tasks:

Action 1: Work-first spawn is performed. $\gamma_{t+1} \in ST_{spawn}(\gamma_t) \subset ST_{spawn}(\gamma_{pt})$.

Action 2: Help-first spawn is performed. $\gamma_{t+1} = \gamma_t \in ST_{spawn}(\gamma_{pt})$.

Action 3: $\gamma_{t+1} \in ST_{sync}(\gamma_t) \subset ST_{spawn}(\gamma_t) \subset ST_{spawn}(\gamma_{pt})$.

Action 4: In this case, $\gamma_{t+1} = PR_{spawn}(\gamma_t)$. Since $\gamma_t \in ST_{spawn}(\gamma_{pt})$ and $\gamma_t \neq \gamma_{pt}$, we have $\gamma_{t+1} \in ST_{spawn}(\gamma_{pt})$.

Action 5: In this case, γ_{t+1} becomes ready before step t+1. This implies γ_{t+1} does not have any live sync tree descendant. Assume $\gamma_{t+1} \notin ST_{spawn}(\gamma_{pt})$. Because $\gamma_t \in ST_{sync}(\gamma_{t+1}) \subset ST_{spawn}(\gamma_{t+1})$ and $\gamma_t \in ST_{spawn}(\gamma_{pt})$, we have $\gamma_{pt} \in ST_{spawn}(\gamma_{t+1})$-$\{\gamma_{t+1}\}$. As we have $\gamma_t \in ST_{spawn}(\gamma_{pt})$ and $PR_{sync}(\gamma_t) = \gamma_{t+1}$, by Property 3.2.2, we have $PR_{sync}(\gamma_{pt}) = \gamma_{t+1}$. This is contradictory to γ_{t+1} has no live sync tree descendant. Thus we have $\gamma_{t+1} \in ST_{spawn}(\gamma_{pt})$.

Idle: If $\gamma_{t+1} = \gamma_{pt}$, $\gamma_{t+1} \in ST_{spawn}(\gamma_{pt})$. If $\gamma_{t+1} \neq \gamma_{pt}$, let γ_c be the common spawn tree ancestor of γ_{pt} and γ_{t+1}. If $\gamma_c = \gamma_{pt}$, then $\gamma_{t+1} \in ST_{spawn}(\gamma_{pt})$. $\gamma_c = \gamma_{t+1}$ is not possible due to tie breaker (c) in the idle routine. The only situation left is γ_c, γ_{pt} and γ_{t+1} are three different tasks.

According to the idle routine, γ_{t+1} is either removed as a preempted task or a fresh task.

If γ_{t+1} is preempted and owned by p before step t+1: because Action 1 is not taken at step t+1, γ_{t+1} must be preempted before step t. Because γ_{pt} and γ_{t+1} are in two disjoint spawn subtree, it is impossible for both of them to make progress due to p at step t. As both γ_{pt} and γ_{t+1} are preempted before step t+1, neither of them is executed at step t. This is contradictory to the assumption argument of Lemma 5.2.2 for step t.

If γ_{t+1} is removed as a fresh task: γ_{pt} cannot be preempted at the same step at which γ_{t+1} is spawned, because fresh task can only be spawned using help-first policy and a task can only be preempted when performing a work-first policy. If γ_{pt} is preempted before γ_{t+1} is spawned, then when γ_{t+1} is spawned, according to the assumption argument of Lemma 5.2.2, $\gamma_q \in ST_{spawn}(\gamma_{pt})$. Thus $\gamma_{t+1} \in ST_{spawn}(\gamma_{pt})$. If γ_{pt} is preempted after γ_{t+1} is spawned using the help-first policy, then when γ_{pt} is preempted, according the assumption argument of

Lemma 5.2.3, we have $\gamma_{pt} \in ST_{sync}(PR_{sync}(\gamma_{t+1}))$. This is not possible due to tie breaker (b) in the idle routine.

All situations are either ruled out or lead to $\gamma_{t+1} \in ST_{spawn}(\gamma_{pt})$ or γ_{pt} be executed at step t+1.

We now show the argument for Lemma 5.2.3 is true by enumerating the situation when Actions 1-5 are taken by the algorithm at step t and the situation when the idle routine is called before step t+1 to remove tasks to execute at step t+1. By assumption, we have $\gamma_t \in ST_{sync}(\gamma_y)$ except for the case for Action 2 when γ_f is spawned at step t.

Action 1: In this case, $\gamma_t = \gamma_q$ and no new fresh task is created. By assumption $\gamma_t \in ST_{sync}(\gamma_y)$. As γ_{t+1} is spawned by γ_t using work-first policy ($\gamma_q = \gamma_t$), by Property 3.2.1, we have either $\gamma_s = \gamma_q = \gamma_t$ or $\gamma_s = PR_{sync}(\gamma_q)$. If $\gamma_s = \gamma_q = \gamma_t$, we have $\gamma_{t+1} \in ST_{sync}(\gamma_s) = ST_{sync}(\gamma_t) \subset ST_{sync}(\gamma_y)$. Otherwise, if $\gamma_s \neq \gamma_q$, we have $\gamma_s = PR_{sync}(\gamma_q)$. Because $\gamma_q = \gamma_t \in ST_{sync}(\gamma_y)$, we have $\gamma_s \in ST_{sync}(\gamma_y)$ unless $\gamma_q = \gamma_y$. If $\gamma_q = \gamma_y$, by Property 3.2.3, because γ_f was already spawned and $PR_{sync}(\gamma_f) = \gamma_y$, we have $PR_{sync}(\gamma_{t+1}) = \gamma_y = \gamma_s$, which is contradictory to $\gamma_s \neq \gamma_q$. So we have $\gamma_s \in ST_{sync}(\gamma_y)$, which implies $\gamma_{t+1} \in ST_{sync}(\gamma_s) \subset ST_{sync}(\gamma_y)$.

Action 2: In this case, we have $\gamma_t = \gamma_{t+1}$. Assume γ_n is the fresh task spawned at this step, we only need to show $\gamma_{t+1} \in ST_{sync}(PR_{sync}(\gamma_n))$ because other fresh tasks are covered by the assumption. Property 3.2.1 leads directly to $\gamma_{t+1} \in ST_{sync}(PR_{sync}(\gamma_n))$.

Action 3: In this case, $\gamma_{t+1} \in ST_{sync}(\gamma_t) \subset ST_{sync}(\gamma_y)$.

Action 4,5: In both cases, γ_t terminates and $\gamma_{t+1} \in ST_{sync}(PR_{sync}(\gamma_t))$ (Property 3.2.1). Because by assumption $\gamma_t \in ST_{sync}(\gamma_y)$, we have $\gamma_{t+1} \in ST_{sync}(\gamma_y)$ unless $\gamma_t = \gamma_y$. It is impossible to have $\gamma_t = \gamma_y$ because γ_f is γ_y's sync tree descendant, which implies $\gamma_y = \gamma_t$ cannot terminates.

Idle: If γ_{t+1} is preempted and owned by p before step $t+1$. Consider the step when γ_{t+1} is preempted and γ_f is spawned. This cannot happen in one step because a task is preempted only at work-first spawn and a fresh task is created only at help-first spawn.

If γ_{t+1} was preempted after γ_f is created, then according to the assumption argument for Lemma 5.2.3 , we have $\gamma_{t+1} \in ST_{sync}(\gamma_y)$.

If γ_{t+1} was preempted before γ_f is created, then according to the assumption for Lemma 5.2.2, we have $\gamma_z \in ST_{spawn}(\gamma_{t+1})$, which implies $\gamma_f \in ST_{spawn}(\gamma_{t+1})$. Since both γ_{t+1} and γ_y are γ_f's spawn tree ancestors, we consider which one is closer to γ_f. If $\gamma_{t+1} = \gamma_y$, then $\gamma_{t+1} \in ST_{sync}(\gamma_y)$. If γ_{t+1} is closer to γ_f than γ_y, by Property 3.2.2, we have $\gamma_y = PR_{sync}(\gamma_{t+1})$ which also implies $\gamma_{t+1} \in ST_{sync}\gamma_y$. If γ_y is closer to γ_f than γ_{t+1}, by Property 3.2.1, we have $PR_{sync}(\gamma_y) \in ST_{sync}(PR_{sync}(\gamma_{t+1}))$. This is not possible because $\gamma_y = PR_{sync}(\gamma_f)$ and the idle routine would return γ_f instead of γ_{t+1} due to tie breaker (a).

If γ_{t+1} is removed as a fresh task, consider the step γ_{t+1} and γ_f are spawned. If $\gamma_{t+1} = \gamma_f$, $\gamma_{t+1} \in ST_{sync}(\gamma_y)$.

If γ_{t+1} was spawned before γ_f, then when γ_f was spawned by γ_z, by assumption for Lemma 5.2.3, we have $\gamma_z \in ST_{sync}(\gamma_s)$. If $\gamma_s = \gamma_z$, by Property 3.2.3, as γ_{t+1} was spawned before γ_f and $PR_{sync}(\gamma_{t+1}) = \gamma_s$, we have $\gamma_y = \gamma_s$. Assume $\gamma_s \neq \gamma_z$. According to Property 3.2.1, we have either $\gamma_z = \gamma_y$ or $PR_{sync}(\gamma_z) = \gamma_y$. $\gamma_z = \gamma_y$ is impossible because, otherwise, γ_f is one level deeper than γ_{t+1} in the sync tree, and should have been removed by the idle routine due to tie breaker (a). If $PR_{sync}(\gamma_z) = \gamma_y$, since we also have $\gamma_z \in ST_{sync}(\gamma_s)$, then we have $\gamma_y \in ST_{sync}(\gamma_s)$ unless $\gamma_z = \gamma_s$. As we assume $\gamma_s \neq \gamma_z$, we have $\gamma_y \in ST_{sync}(\gamma_s)$. This implies $\gamma_y = \gamma_s$, because otherwise, γ_f is one level deeper than γ_{t+1} in the sync tree, and should have been removed by the idle routine due to tie breaker (a). All cases lead to $\gamma_y = \gamma_s$, which leads to $\gamma_{t+1} \in ST_{sync}(\gamma_s) = ST_{sync}(\gamma_y)$.

If γ_{t+1} was spawned after γ_f, then when γ_{t+1} was spawned by γ_q, by assumption for Lemma 5.2.3, we have $\gamma_q \in ST_{sync}(\gamma_y)$. By Property 3.2.1, either $\gamma_q = \gamma_s$ or $PR_{sync}(\gamma_q) = \gamma_s$. If $\gamma_q = \gamma_s$, we have $\gamma_s = \gamma_q \in ST_{sync}(\gamma_y)$. Thus, $\gamma_{t+1} \in ST_{sync}(\gamma_s) \subset ST_{sync}(\gamma_y)$. If $PR_{sync}(\gamma_q) = \gamma_s$, because we already have $\gamma_q \in ST_{sync}(\gamma_y)$, we have $\gamma_s \in ST_{sync}(\gamma_y)$ unless $\gamma_q = \gamma_y$. If $\gamma_s \in ST_{sync}(\gamma_y)$, we have $\gamma_{t+1} \in ST_{sync}(\gamma_s) \subset ST_{sync}(\gamma_y)$. If $\gamma_q = \gamma_y$, according to Property 3.2.3, as γ_f was spawned before γ_{t+1} and $PR_{sync}(\gamma_f) = \gamma_y$, we have $\gamma_s = \gamma_y$. This will also lead to $\gamma_{t+1} \in ST_{sync}(\gamma_s) = ST_{sync}(\gamma_y)$.

All situations are either ruled out or lead to $\gamma_{t+1} \in ST_{sync}(\gamma_y)$. □ □

Theorem 5.2.4. *All adaptive schedules are progressive.*

Proof. For any non-fresh task γ: if γ is executed by some processor, it is making progress; if γ is preempted, it is making progress by Lemma 5.2.2; if γ is suspended, γ must have live descendants in the sync sub tree $ST_{sync}(\gamma)$. Assume γ_a is one of leaves in $ST_{sync}(\gamma)$. γ_a is not suspended because it is a leaf. If γ_a is preempted or being executed by some processor, γ_a is making progress due to Lemma 5.2.2, which implies γ is also making progress. If γ_a is fresh, γ is making progress due to Lemma 5.2.3. □ □

5.2.3 Theoretical Space Bound

Given a dag $G(V, E, S_1)$, the following theorem presents the space bound for any P-processor adaptive schedule.

Theorem 5.2.5. *If $S_1 <= S$, the memory space of any $P - ADP(S, F)$ schedule is bounded by $S_1 P + O(FP)$. If $S_1 > S$, the memory space of any $P - ADP(S, F)$ schedule is bounded by $S_1 P + O(V)$.*

Proof. According the Theorem 5.2.4, adaptive scheduler is progressive. Thus, the memory bound for non-fresh tasks is $S_1 P$ according to Theorem 5.2.1. We calculate

Algorithm 1 Adaptive Work-Stealing Algorithm - ADP(S,F)

Environment: There are P processors and a shared task pool where every processor can remove and put tasks. All operations are assumed to be atomic.

The algorithm proceeds step by step. Note that both the spawn tree and the sync tree are unfolded online as the algorithm progresses.

When a processor p is idle before step t, it will call the *idle routine* in routine 1 to attempt to remove and execute task for the step t.

Step 0: At step 0, one processor will start executing the root task. All other processors are idle.

Step t+1: For step t, the task will decide the task to execute for step t+1. If processor p executes task γ_a at step t, it will execute the next instruction in task γ_a unless γ_a spawns, suspended or terminates. In these cases, the following rules are followed:

Spawn: Let γ_a spawns γ_b. Processor p will use the following rule to decide the spawn policy:

1. If the space of the activation frames of all tasks marked *on-stack-p* $\geq S$, use help-first (**stack condition**);
2. Otherwise if the number of fresh tasks currently owned by p before t is $\geq F$, use work-first (**fresh-task condition**);
3. Otherwise free to use any heuristic. See Section 5.2 for SLAW's heuristic.

Action 1: If the spawn is under work-first policy, return γ_a to the pool and execute γ_b for step t+1.

Action 2: If the spawn is under help-first policy, put γ_b to the pool and continue to execute next instruction of γ_a for step t+1.

Suspended: If the task γ_a is suspended, processor p will return γ_a to the pool, do a context switch and clears all *on-stack-p* flags on tasks. Then

Action 3: processor p will remove any fresh task it created in $ST_{sync}(\gamma_a)$. If not success, p becomes idle.

Terminates: If the task γ_a terminates and if γ_a is the root task, then the schedule ends. Otherwise, let T_1 be $PR_{spawn}(\gamma_a)$ and T_n be $PR_{sync}(\gamma_a)$. Processor p will:

Action 4: If T_1 is preempted and owned by p, remove T_1 and execute it for step t.

If Action 4 is not taken, the processor p will do a context switch and clears all *on-stack-p* flags on tasks. Then it will

Action 5: Check if T_n becomes ready before step $t+1$ If yes, processor p will attempt to remove T_n from the pool and execute T_n for step $t+1$. If the Action 5 is not taken, processor p will becomes idle.

---

the heap space used by fresh tasks in each cases. If $S_1 <= S$, the stack condition in Algorithm 1 will never be met. Thus, the number of fresh task is bounded by F per processor according to the fresh task condition. The total heap space for fresh task is bounded by $O(FP)$. If $S_1 > S$, the bound V for the number of fresh tasks is trivial before it is the total number tasks in the dag. This completes the proof. □ □

Theorem 5.2.5 establishes the memory space bound for the adaptive schedule. If $S_1 <= S$, this is the case that work-first can run successfully without exceeding the stack bound. The work-first work-stealing scheduler's memory bound in this case is S_1P. The memory space of the adaptive scheduler is bounded by $S_1P + O(FP)$. If $S_1 > S$, this is the case that work-first will overflow the stack. The adaptive schedule will never overflow the stack and the memory space is bounded by $S_1P + O(V)$.

Routine 1 Idle Routine for Processor p Before Step t

1. If p owns any fresh task or preempted task, remove one. If multiple such tasks exist, the following tie breaker is used:

   (a) Return one that is the deepest in sync tree.

   (b) Return preempted tasks before fresh task

   (c) Return one that is the deepest in spawn tree

   If success, goto 4.

2. In this case, processor p does not own any task. It will go stealing. It will attempt to remove task γ in the pool that meets one of the following stealing restrictions.

   (a) if γ is fresh and created by processor some q, γ is the one that was created earliest among all fresh tasks created by q.

   (b) if γ is preempted and owned by processor some q, γ is the one that was preempted earliest among all tasks preempted and owned by q.

   If success, goto 4.

3. Processor p remains idle. Goto 1.

4. Processor p returns the task for execution at step t.

It is importance to notice that the S_1 is related to the input data size [14] because it is the space requirement of serial depth-first execution. However, F is a preset parameter and is not related to the input data size. The constant of the $O(FP)$ is the size of the holder of the fresh task on the heap, which is usually small. In SLAW, the task holder contains only the value of input parameters to the task function and a few bookkeeping fields.

5.2.4 Runtime Implementation

SLAW implements the adaptive scheduling algorithm using two deques per worker: one for preempted tasks owned by the worker and the other for fresh tasks created by the worker. When a task is preempted at a work-first spawn, it is pushed to the bottom-end of the preempted task deque. When a fresh task is created using the help-first policy, it is pushed to the bottom-end of the fresh task deque. When a thief is stealing, it steals from the top-end of either one of the other workers' deques. When looking for the preempted task that is the deepest in the task spawn tree, one need only to check the bottommost frame from the preempted task deque. When looking for the fresh task that is the deepest in the task sync tree, one need only to check the bottommost frame from the fresh task deque. This simplifies the implementation of Action 4 and the tie breakers in the idle routine. When task Γ_a terminates, if $T_1 = PR_{spawn}(\Gamma_a)$ is preempted and owned by the current worker, Γ_a must have been spawned under the work-first policy by Γ_a. To take Action 4 and execute T_1, the worker needs to simply return from the function and pop the bottommost frame of the preempted and owned task deque. If Action 4 is not taken, then the worker will check if the $T_n = PR_{sync})(\Gamma_a)$ is ready (line 30-34 in Figure 4.20). If not ready, then either a task in $ST_{sync}(T_n)$ is returned at line 31, or another task is returned at line 37. The task returned is selected according to the tie breaker of the idle routine and the process only involves peeking at the bottommost frames from both deques.

5.3 Experimental Results

5.3.1 Setup

performance results are obtained on the following two machines:

1. **Niagara 2**: This system includes a 8-core 64-thread 1.2GHz UltraSPARC T2 processor with 32GB main memory. All cores share a single 4MB L2 cache, thus it is not interesting to locality-aware scheduling. Only locality-oblivious

deployment is used (1 place for all workers) on Niagara 2.

2. **Xeon SMP**: This system includes four Quad-Core Intel E7330 processors running at 2.40GHz with 32GB main memory. Each Quad-core processor has two core-pairs and each core-pair share a 3MB L2 cache. The locality-aware deployment provided for the SLAW scheduler has 8 places, with 1 or 2 workers per place.

The implementation used to evaluate SLAW in this paper is based on Java to facilitate portability across the above systems. Each worker is implemented as a separate Java thread. The JVM used on both machines is Sun Hotspot JDK 1.6. In both cases, the JVM was invoked with the following parameters: "-Xmx2g -Xms2g -Xmn1g -server -Xss8M -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:+UseBiasedLocking -XX:+AggressiveOpts". The experiment also includes some Cilk++ results. The Cilk++ release used is based on gcc v4.2.4. Both Cilk++ code and the serial C code were compiled using the -O2 option.

We evaluate the SLAW work-stealing scheduler on a variety of benchmarks listed in Table 5.1. To reduce the impact of JVM overheads in the evaluation, including JIT compilation and garbage collection, the execution time reported is the average of the three best benchmark iterations from three separate VM invocations. Each VM invocation performs 10 benchmark iterations.

5.3.2 Sensitivity Analysis of Parameters in SLAW Scheduler

Figures 5.2(a) - 5.2(f) contain performance results obtained on the Niagara 2 machine to analyze the sensitivity of the INT, F, and S parameters on the performance of the SLAW scheduler, as discussed in the following subsections. Based on this sensitivity analysis, the default parameter value of SLAW is presented in Table 5.2.

Impact of INT Parameter

Figures 5.2(a) - 5.2(d) study the impact of the policy evaluation interval, INT, on the performance of the SLAW scheduler.

Figure 5.2(a) shows the impact of INT on the execution time of the Fib(35) microbenchmark on 1 worker, with S and F set to their default values of 256 and 128 respectively. Since no stealing occurs in the 1-worker case, the adaptive heuristic will switch very quickly from help-first to work-first, and each subsequent re-evaluation will keep the policy as work-first for this case. The largest overhead is incurred when INT=1, since the spawning policy is re-evaluated for every spawned task. This suggests that INT should not be made too small. However, even in the INT=1 worst case, the overhead of the adaptive policy is only about 5% compared to the work-first policy. The overhead rapidly approaches zero with increasing values of INT. The execution time for the help-first policy is too big to fit in Figure 5.2(a) (about 9× slower than the work-first policy due to the context switching incurred at every task synchronization point).

Figure 5.2(b) repeats the evaluation in Figure 5.2(a), but with 32 workers instead of 1 worker. In this case, we see a negative performance impact of selecting an INT value that's too large. If the interval is too large, the performance degrades as shown in Figure 5.2(b) and 5.2(d). This is because, for a recursive benchmark like Fib, work-first is the best spawning policy. The SLAW scheduler will start with the help-first policy at the beginning and then switch to work-first. However, if INT is too large, then some noticeable context switch overhead will be observed before the policy switch occurs. The same situation occurs for the PDFS benchmark in Figure 5.2(d). When we increase the INT value past 64, the throughput of the parallel depth first search benchmark declines.

Figure 5.2(c) shows the impact of INT on the execution time of SOR on 64 workers. In this fork-join version of SOR, 64 tasks are distributed among 64 workers in each outer (time-step) iteration, and these tasks are joined with a finish construct at the

end of each iteration. Note that stealing is very frequent in this example, since 63 out of 64 tasks will be stolen in each iteration, thereby implying that the stealing rate is high and help-first policy is the best choice. . Because the adaptive schedule starts with help-first policy and re-evaluates the policy after every INT spawns, this experiment suggests that the INT should be set to be greater or equal the number of workers. Doing so will ensure that at least one task is spawned for each worker using the help-first policy before the worker switches to work-first policy. If a worker switches to the work-first policy too early, it will delay task creation and negatively affect the performance of the entire application. For the same reason, the fresh deque threshold should also be to be equal or greater than the number of workers to hold at least one task for each worker.

For the reasons described above, since the maximum number of workers is 64 (on Niagara 2), the default value of INT is set to 64 for the experiments presented in this paper.

Impact of Parameter F

Figure 5.2(e) shows how the throughput of the PDFS benchmark varies for different values of the fresh task threshold, F. The other two parameters INT and S are fixed at 50 and 256 respectively. INT is fixed at 50 for this example because (INT=50,S=256) was the best combination we found for PDFS after enumerating the parameter space. All other experimental results reported in this paper use the default parameter value unless otherwise specified.

We do not find any correlation between the throughput and F. This is in part because F is a soft bound that has lower priority in the SLAW algorithm than the stack bound. The worker will always use the help-first policy to create fresh tasks regardless of the number of existing fresh tasks, if the stack bound prevents the use of the work-first policy.

In the previous discussion on the parameter INT, we also mentioned the reason

why F should be equal or greater than the number of workers. The default value of F in SLAW is set to 128.

Impact of Parameter S

Figure 5.2(f) shows the throughput of the PDFS benchmark as a function of the stack threshold S. When S is set to 1, the adaptive schedule becomes equivalent to the work-first schedule. Interestingly, if the stack threshold is set too large, the performance also degrades. This is because if the stack becomes too deep, the number of memory pages spanned by the runtime call stack increases, which in turn leads to an increase in TLB misses. The default stack threshold in SLAW is set to 256 activation frames. Among the benchmarks used in this paper, only the PDFS benchmark requires a stack that grows proportionally with the input problem size. The stack requirement for other benchmarks is bounded by a small number; consequently they do not hit the stack bound.

5.3.3 Benchmark Results

One optimization that has been studied in previous research is to transform some flat loops to recursive style [48, 9]. This optimization requires compiler support, and only applies to do-all loops on a divisible region and does not apply to do-cross loops or pointer-chasing programs. As the main contribution of this paper is to show the robustness of the runtime, we do not apply such optimizations. However, for the FJ microbenchmark, we do show the performance results of with and without the recursive loop optimization.

We use two microbenchmarks, Fib and FJ, to show the extreme cases where work-first policy is better than help-first policy and the help-first policy is better than the work-first policy respectively. Fib is used as the extreme case for recursive parallelism and FJ without the recursive loop optimization is used as the extreme case for flat parallelism. Table 5.3 shows the execution time of Fib on Niagara 2. Figure 5.3

shows the number of fork-joins performed per second. For Fib, the work-first policy is 10.2× faster than the help-first policy due to fewer context switches. In FJ, steals are frequent and the help-first policy is 4.6× faster than the work-first policy. In both micro-benchmarks, the performance of the adaptive scheduler is close to that of the better policy.

Figure 5.4 shows the number of fork-joins performed per second in a recursive-style fork-join (`fj-rec`), and compares the number to the iterative fork-join (`fj`). In `fj-rec`, the tasks are recursively spawned. In order to spawn 1024 parallel tasks, the depth of the task spawn tree is 11. When the number of threads is small (<= 8), the work-first policy performs than better than the help-first policy. This is because the number of steals is infrequent compared to the task spawned. As the number of threads increases, the parallelism in the program becomes insufficient and the steal becomes more frequent (considering the depth of the task spawn is 10 for 1024 tasks). This explains why the help-first policy performs better than the work-first policy as the number of threads increases. This example is interesting because it shows that the best choice of scheduling policy is more a dynamic choice than a static choice, although the shape of a program can probably give some clue. The experiment also confirms that `fj-rec` is more scalable than `fj`, as the task spawns are now performed in parallel as well. However, the sequential overhead of the `fj-rec` is higher than the iterative `fj`, which explains the lower performance when the number of threads is small.

Figure 5.5 shows the speedup of the SLAW scheduler on Niagara 2 over the Java-serial version with one exception for PDFS, whose speedup is based on 1-thread help-first execution. Both the serial version and the work-first schedule of PDFS will overflow the stack as described in Section 5.1.4. This is why there is no bar for the *wf* in the figure. This exception also applies to Figure 5.6.

Three scheduling policies are compared: help-first only, work-first only and the adaptive scheduling algorithm described in Section 5.2. As all cores on Niagara 2

share the same L2 cache, this experiment uses the locality-oblivious deployment, which specifies only 1 place with all 64 workers. No processor binding is used.

CG.A, MG.A and SOR are flat, loop-based parallel benchmarks. In these benchmarks, the help-first policy performs better than the work-first policy. The results in Figure 5.5 show that the adaptive scheduling algorithm matches or exceeds the performance of the help-first policy for these benchmarks.

Sort, Matmul, LU and GC are task recursive parallel benchmarks. In these benchmarks, the work-first policy is better than or almost the same as the help-first policy. The result shows the adaptive scheduling algorithm matches or exceeds the performance of work-first policy in those benchmarks.

PDFS is an irregular graph computation. Irregular graph computations are interesting because the structure of the spawn tree depends on the order in which nodes are visited (labeled) in parallel. We used the Parallel Depth First Search benchmark (PDFS) studied in [23] (kernel code shown in Figure 4.2), and applied it to a two-dimensional 2000×2000 torus graph consisting of 4 million nodes. Our results show that the adaptive approach outperformed the help-first policy for this benchmark because of its ability to combine help-first and work-first policies. At the beginning, all workers are idle and stealing is frequent thereby making help-first the more desirable policy. After each worker gets some work, they begin traversing the graph and stealing becomes less frequent, thus causing the adaptive runtime to switch to the work-first policy. The work-first policy incurs no synchronization overhead and executes the tasks as if they are sequential calls. Finally, according to the stack condition in the adaptive scheduling algorithm, the runtime will switch back to the help-first policy when it becomes necessary to avoid overflowing the stack size limit.

Figure 5.6 shows the speedup of the SLAW scheduler on Xeon SMP using the three scheduling policies. For reference, we report also Cilk++'s speedup for those benchmarks for which Cilk version is available(SORT, MATMUL, LU). We also

translate the JGF SOR to Cilk++ and use the *cilk_for* to parallelize the loop. To factor out uniprocessor performance differences between Java and C, the speedup for SLAW in this figure is based on the Java-serial version and Cilk++'s speedup is based on the C-serial version. This experiment uses the locality-oblivious deployment. The experimental results of the locality-aware scheduling are presented later in the Section 6.2.

For Sort, Matmul and LU, SLAW achieves over 10× speedup on Xeon SMP. SLAW scales almost linearly on Matmul. Cilk++ also scales almost linearly from 1 worker to 16 workers, but its speedup looks smaller because Cilk++'s 1-worker case is 2.4× slower than the C-serial version due to some optimizations that are disabled by the Cilk++ compiler. CG, MG and SOR do not scale beyond 4× hit a memory-bandwidth wall. The scalability of CG and SOR can be significantly improved by locality-aware scheduling as shown next in Section 6.2.

5.3.4 Modeling and Measurement of Overhead

In this section, we model and measure the overhead of context switches, asynchronous function calls and task synchronizations using the iterative fork-join and the recursive fib micro-benchmarks.

According to the work-first principle, the scheduler can achieve almost linear speedup. We shall optimize the single thread execution time even at the cost of increasing the critical path.

We breakdown the single thread execution (t_1) of a HJ program into the following components: the serial Java execution (t_s), the asynchronous task spawns, context switches (t_{cs}) before starting a new task, `startFinish` (t_{sf}) and task synchronization at `endFinish`. Depending on the scheduling policy, the overhead of asynchronous task spawns is denoted as t_{aw} for work-first task spawns or t_{ah} for help-first task spawns. The task synchronization at `endFinish` is either trivial or non-trivial. In the trivial case, all tasks created in the finish scope are completed and the worker will

just continue execution. In the non-trivial case, the current serial execution flow will be interrupted (an exception is thrown and caught by the runtime), and a context switch is performed before executing new tasks. We use t_{ef} to denote the overhead of non-trivial task synchronization. The overhead of the trivial case is considered to be included in t_{sf}.

Consider the HJ program shown in Figure 5.7. The program performs k tasks; task 1 to $k-1$ are performed asynchronously and task 0 is performed serially. Under the work-first policy, the single thread execution incurs no context switch. Thus, the single thread execution time of the whole program as a function of k for the work-first policy is

$$t_1^{wf}(k) = t_s(k) + t_{sf} + (k-1)t_{aw} \tag{5.1}$$

where $t_s(k)$ is the serial execution time of the whole program as a function of k. There is no extra synchronization cost (t_{ef}) at the end of the finish scope for single-thread work-first execution.

Under the help-first policy, task 1 to $k-1$ will be executed after a context switch and there is one non-trivial task synchronization at `end-finish` for $k > 1$. Thus

$$t_1^{hf}(k) = \begin{cases} t_s(k) + t_{sf}, & k = 1 \\ t_s(k) + t_{sf} + (k-1)(t_{ah} + t_{cs}) + t_{ef}, & k > 1 \end{cases} \tag{5.2}$$

We assume every task in the program shown in Figure 5.7 contains the same amount of work which is also called task granularity (t_0). The task granularity t_0 can be calculated as the slope of the serial execution $t_s(k)$.

Based Equation 5.1 and 5.2, we have

$$t_{sf} = t_1^{wf}(1) - t_s = t_1^{hf}(1) - t_s(1)$$

$$t_{aw} = SLOPE(k, t_1^{wf}(k)) - t_0 \quad (k >= 1)$$

$$t_{ah} + t_{cs} = SLOPE(k, t_1^{hf}(k)) - t_0 \quad (k > 1)$$

$$t_{ef} = t_1^{hf}(2) - t_1^{hf}(1) - t_{aw} + t_{cs} - t_0$$

.

Table 5.4 shows the execution time of the serial Java execution time and the single thread HJ execution time of the program in Figure 5.7 with $k = 1, 2, 4, 8, ..., 1024$ on a Xeon SMP machine.

We calculate t_0, t_{sf}, t_{aw}, $t_{ah} + t_{cs}$ and t_{ef} and get the following result: $t_0 \approx 0.1\mu s$, or 417 cycles, $t_{sf} \approx 0.1\mu s$ or 417 cycles, $t_{aw} \approx 0.15\mu s$ or 625 cycles, $t_{ah} + t_{cs} \approx 0.22\mu s$ or 917 cycles, $t_{ef} \approx 2.11\mu s$ or 8800 cycles.

Consider the HJ program shown in Figure 5.8 and Figure 5.9. The code shown in Figure 5.8 uses a global finish to synchronize all task where as each task in the code shown in Figure 5.9 synchronizes all child tasks. For fib(35), both code spawns $29,860,703$ tasks and the code shown in Figure 5.9 will create $14,930,351$ finish instances. Using the definitions and notations used in the previous example, the 1-thread execution time of the code shown in Figure 5.8 under the work-first and help-first policy are:

$t_{wf}^1 = t_s + t_{sf} + t_{aw} * 29,860,703$

$t_{hf}^1 = t_s + t_{sf} + t_{ef} + (t_{ah} + t_{cs}) * 29,860,703$

The 1-thread execution time of the code shown in Figure 5.9 under the work-first and help-first policy are:

$t_{wf}^2 = t_s + t_{sf} * 14,930,351 + t_{aw} * 29,860,703$

$t_{hf}^2 = t_s + (t_{sf} + t_{ef}) * 14,930,351 + (t_{ah} + t_{cs}) * 29,860,703$

Table 5.5 shows the serial and 1-thread execution time of the code shown in Figure 5.8 and 5.9 under both policies.

We compute t_{aw}, $t_{ah} + t_{cs}$, t_{sf} and t_{ef}, and get the following result: $t_{aw} = 0.11\mu s$ (460 cycles), $t_{ah} + t_{cs} = 0.23\mu s$ (960 cycles),$t_{sf} = 0.17\mu s$ (709 cycles), $t_{ef} = 2.26\mu s$ (9400 cycles).

The results from both examples confirm that spawning and executing a task under the help-first policy (about 0.22-0.23 microseconds) is slower than the work-first policy

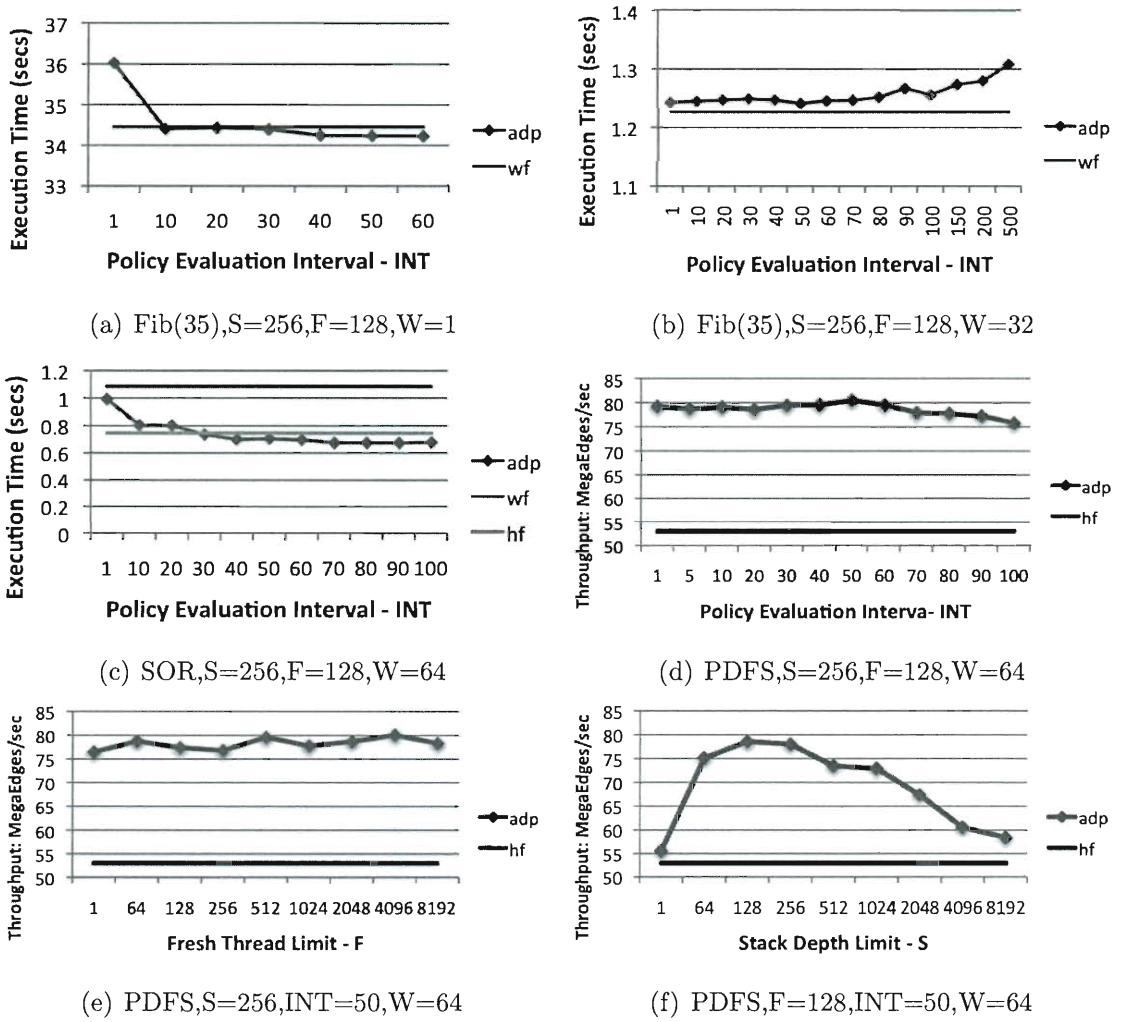
(about 0.11-0.15 microseconds) , and the context switch at the `endFinish` instruction is the most expensive operation (about 2.11-2.26 microseconds). The construction and destruction of a finish instance costs about 0.10-0.17 microseconds.

Benchmark	Type	Description	Source
Fib(35)	Micro Recursive	Recursive Fibonacci (n=35) no sequential threshold/cutoff	Cilk++
FJ(1024)	Micro Flat	Spawn and join 1024 dummy tasks	JGF
SOR	Loop	2D Successive Over-Relaxation algorithm on a 2000 × 2000 float array	JGF
CG.A	Loop	Conjugate Gradient, size A	NPB 3.0
MG.A	Loop	Multi-Grid, size A	NPB 3.0
Sort	Recursive	Parallel Merge Sort on 50331648 random integers	BOTS [40]
Matmul	Recursive	Recursive Matrix Multiplication. (two 1500*1500 double matrix , Threshold=64)	Cilk++
LU	Recursive	Recursive LU Decomposition (2048*2048 double matrix, BlockSize=64)	JCilk
GC	Recursive	Graph Coloring using Parallel Constraint Satisfaction Search (CLIQUE_10,10 colors)	[43]
PDFS	Irregular Recursive	Parallel-DFS(Figure 4.2) on a Torus graph with 4M nodes	XWS [23]

Table 5.1 : List of benchmarks implemented in HJ and their sources

Parameter	INT	F	S
Default Value	64	128	256 activation frames

Table 5.2 : Default Adaptive Schedule Parameters Value



(a) Fib(35),S=256,F=128,W=1

(b) Fib(35),S=256,F=128,W=32

(c) SOR,S=256,F=128,W=64

(d) PDFS,S=256,F=128,W=64

(e) PDFS,S=256,INT=50,W=64

(f) PDFS,F=128,INT=50,W=64

Figure 5.2 : Analysis of Adaptive Schedule Parameter Sensitivity on the Niagara 2 system. The benchmark name, SLAW parameter values (S, F, INT), and the number of workers (W) are specified in the sub-figure captions. Better performance is indicated by smaller values in (a),(b),(c) and larger values in (d),(e),(f).

Wrks	1	2	4	8	16	32	64
hf	334.14	173.64	79.43	39.71	21.43	11.04	8.04
wf	34.45	17.13	8.65	4.31	2.24	1.23	0.87
adp	34.25	16.99	8.54	4.36	2.25	1.25	0.90

Table 5.3 : Performance results for Fib(35) microbenchmark on Niagara 2 using 1 to 64 workers. Execution time (in seconds) is reported. (Smaller is better.)

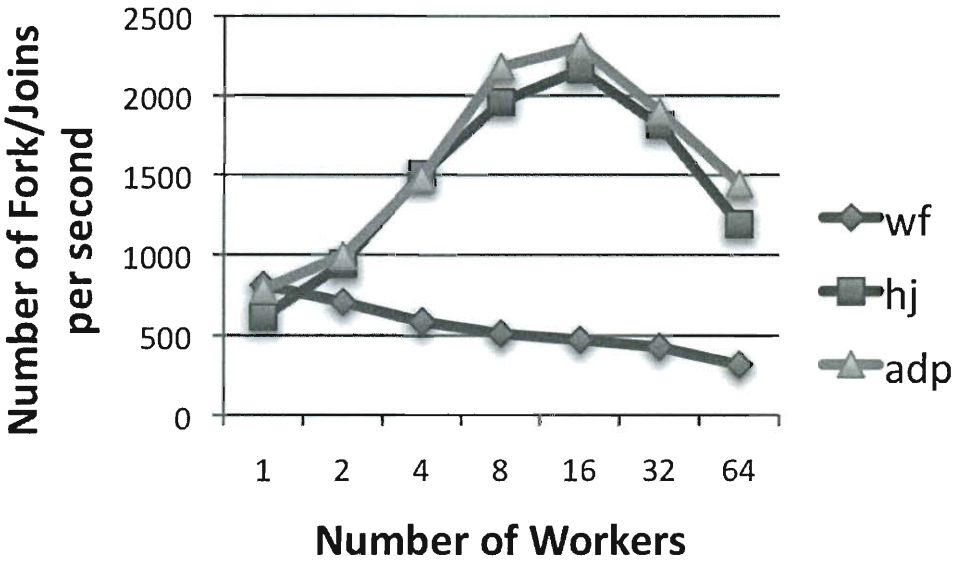


Figure 5.3 : Performance results for FJ(1024) microbenchmark (tasks are spawned iteratively) on Niagara 2 using 1 to 64 workers. Number of fork-joins performed per second is reported. (Bigger is better.)

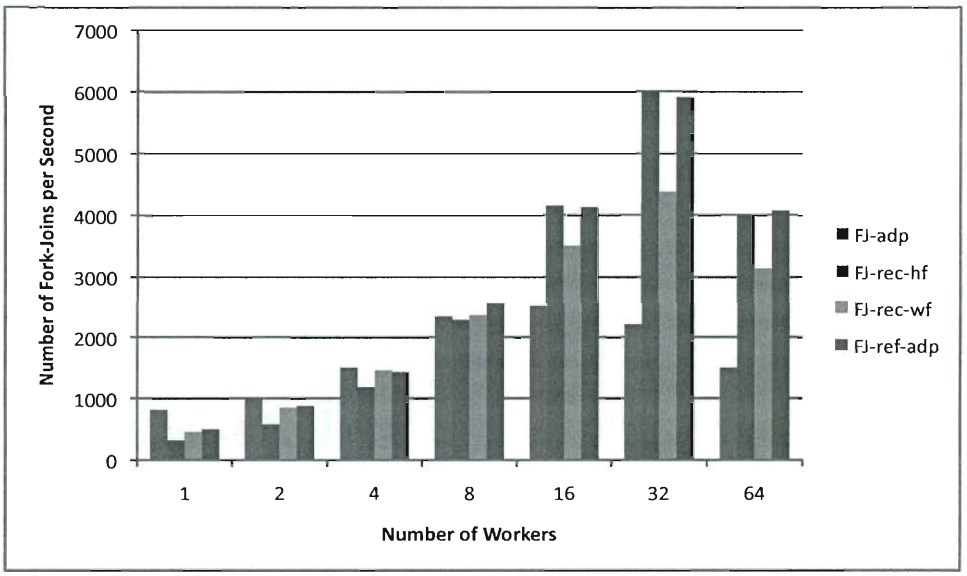


Figure 5.4 : Performance results for FJ(1024) microbenchmark (in FJ-rec, tasks are spawned recursively.) on Niagara 2 using 1 to 64 workers. Number of fork-joins performed per second is reported. (Bigger is better.)

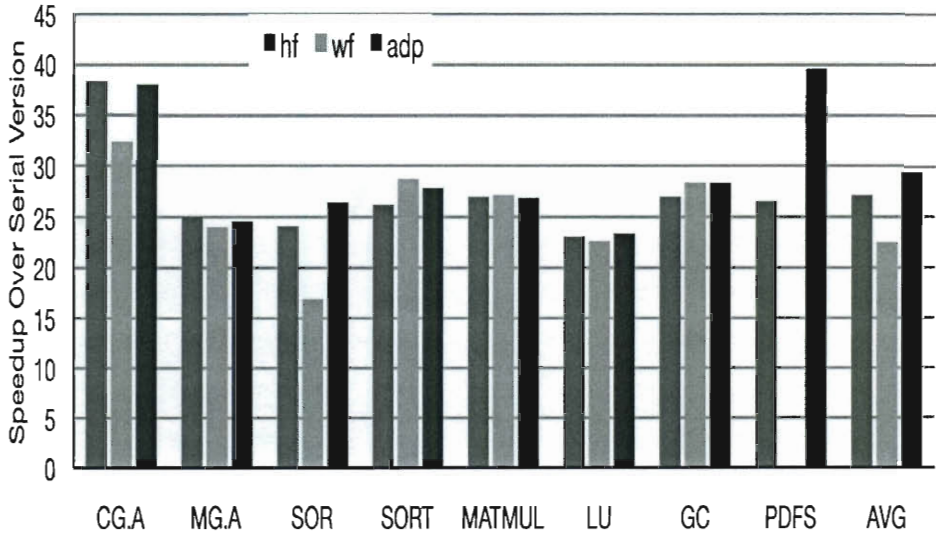


Figure 5.5 : Performance results on Niagara 2. Deployment is locality-oblivious(1-place, 64 workers) with no processor binding.

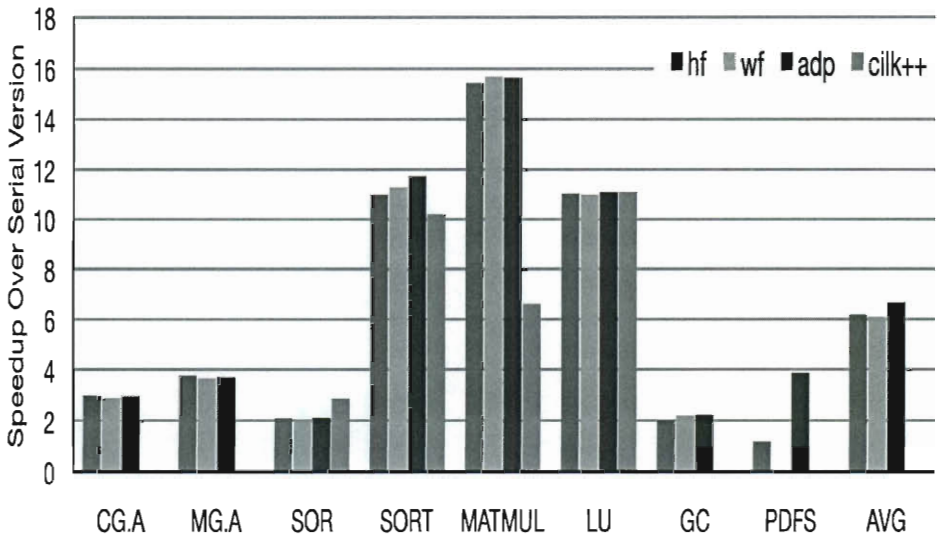


Figure 5.6 : Performance results on Xeon SMP. Deployment is locality-oblivious.(1-place, 16 workers) with no processor binding.

```
finish { //startFinish
     for (int i=1; i<k; i++)
           async Ti;  // task i
     T0; //task 0
}
```

Figure 5.7 : Iterative Fork-Join Example

k	$t_s(k)$	$t_1^{wf}(k)$	$t_1^{hf}(k)$
1	0.11206055	0.207397452	0.218383764
2	0.219238292	0.437988302	2.801269535
4	0.444824225	0.883300931	2.946532226
8	0.899048537	1.95532474	3.916515554
16	1.801026225	3.794922394	6.280302459
32	3.595712472	7.152359563	10.36623923
64	7.174836414	14.58597704	19.61168857
128	14.47073294	28.33663927	36.30554749
256	28.92932566	56.75046819	73.1635938
512	57.53077897	114.1161703	148.6104919
1024	114.8501206	270.4164413	347.826087

Table 5.4 : Execution time (in microseconds) of the serial execution time and the single thread HJ execution time of the program in Figure 5.7 with $k = 1, 2, 4, 8, ..., 1024$ on the Xeon SMP machine

	serial	1-thread work-first	1-thread help-first
Code in Figure 5.8	0.103	3.405	6.974
Code in Figure 5.9	0.103	5.872	43.18

Table 5.5 : Execution time (in secs) of the serial and 1-thread execution time of the code shown in Figure 5.8 and 5.9 under both policies. Both code has the same serial version.

```
finish fib(n);

fib (int n) {
    if (n<2) {
        sum.send(n);
    } else {
         async fib(n-1);
         async fib(n-2);
     }
}
```

Figure 5.8 : Recursive Fib benchmark with one global finish scope.

```
void fib (int n) {
    if (n<2) {
        sum.send(n)
    } else {
         finish {
              async fib(n-1);
             async fib(n-2);
         }
     }
}
```

Figure 5.9 : Recursive Fib benchmark in which every task synchronizes its child tasks

Chapter 6

Locality-aware Work-stealing

6.1 Locality-aware Framework

This section assumes a flat partitioned-global-address space (PGAS) model for locality-aware task scheduling. This model has been adopted by multiple parallel programming languages including Unified Parallel C [33], Co-Array Fortran [68], X10 [20] and the recent release of HJ. We defer the discussion of recent work on the Hierarchical Place Tree to Section 6.3.

The PGAS model in Habanero-Java is derived from the concept of places in X10 1.5. The number of places is a runtime constant specified by the runtime deployment. In X10 1.5, objects and tasks once created in a particular place will be confined to that place, and accessing data in other places will result in `BadPlaceException`. This design is natural for clusters with distributed memory systems. Since most current multi-core architectures are hardware-controlled shared memory systems, places in recent release of HJ only serve as a locality hint for tasks not for data. HJ runtime is free to schedule tasks with no restrictions. The locality-aware framework is an approach to improve performance by utilizing the locality hints provided by programmers.

Figure 6.1 shows the framework of locality-aware scheduling. Each place can contain multiple workers, and has a *mailbox* to store incoming tasks sent from remote places. When a task is spawned using an async statement in HJ, the programmer can specify a locality hint for the task. If the locality hint is not specified, it is understood to be *here* by default, which is the place of the current worker of the parent task. A task is considered local if the locality hint matches the current place, otherwise, it is

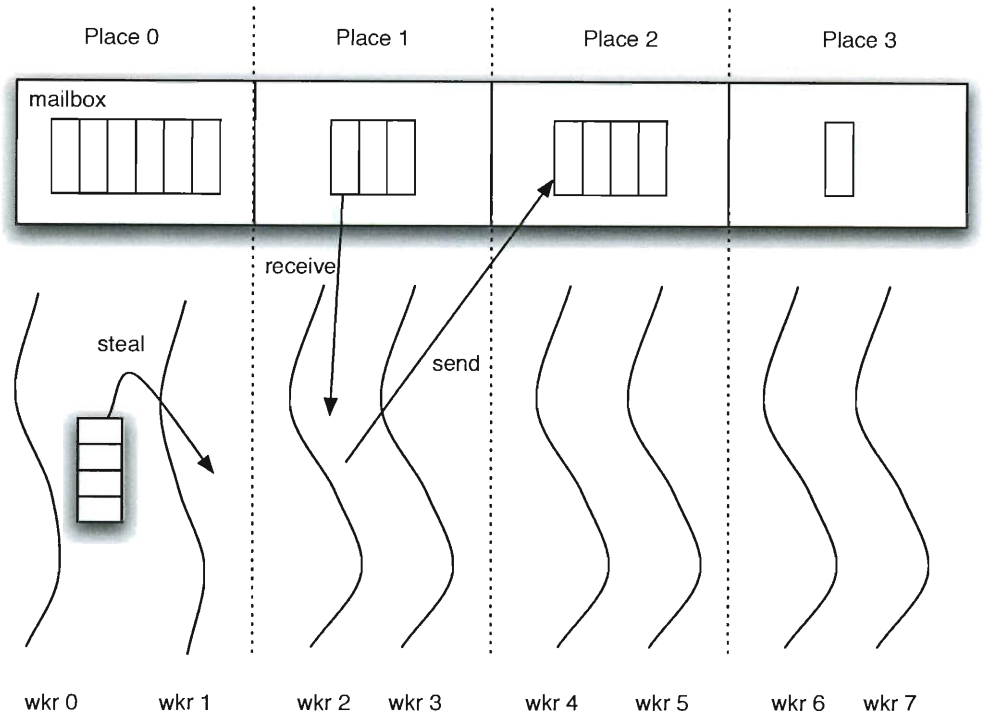


Figure 6.1 : Locality-aware Scheduling Framework

called remote task.

The compiler generates code to test if a task is local or remote when an async statement is encountered. If the task is remote, it will be sent to the mailbox of the place that matches the locality hint, unless the mailbox is full. If the mailbox is full or the task is local, the task will be scheduled according to the policy chosen by the programmer or the runtime as discussed earlier in the dissertation.

When a worker is out of work on local deques, it will try to steal work elsewhere. Under the locality-aware framework, the idle worker will look for work in the following order:

1. the mailbox of the current place.

2. the deques of the peer workers in the same place.

3. the mailboxes of remote places.

4. the deques of workers in remote places.

SLAW has an option to disable cross-place stealing by not allowing the worker to get tasks from remote places (option 3, 4). Whether cross-place stealing is allowed is a tradeoff between cross-place load imbalance and the penalty of counter-productive steals.

6.2 Case Study

In this section, we show how to use locality hints (place clause) to improve the performance of a Habanero-Java application. Especially, we show how to increase temporal cache data reuse for iterative data-parallel applications.

SOR is an iterative data-parallel application from Java Grande Forum Benchmark suite. The original data set is a 2000×2000 double-precision matrix, the size of which is about 32M. The whole data set is too large to fit into the caches of the experimental machine which has only 24M total cache. So for demonstration purpose, we change it real number to single-precision, making the total size of the data set about 16M.

The locality-aware experiments are performed on the Xeon SMP described in Section 5.3.1. With 4 quad-core processors, there are in total of eight(8) L2 shared caches on the machine with a total size of 24M. Thus, the locality-aware deployment provided for the SLAW scheduler specifies 8-places, with 1 or 2 workers per place. The SLAW scheduler will bind workers to virtual processors when the worker threads are created.

Figure 6.2 shows the task spawn tree in two consecutive iterations for 8 places with 2 workers per place. Between two iterations, the task in the rectangular will access the same range of data. Two adjacent tasks will be executed by workers in the same place. Divided by 8 places, the data for each place (approximately 2MB) fits into the 3M L2 cache.

Under randomized work-stealing, cache misses can occur when a task migrates from one worker to another, when the two workers execute on cores that access

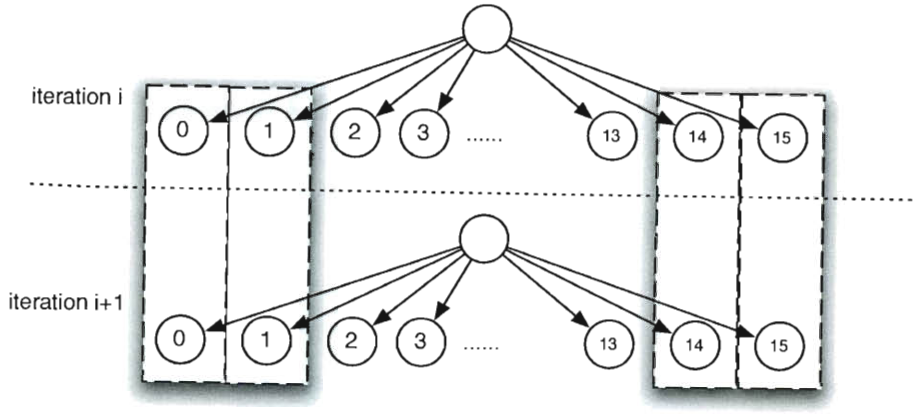


Figure 6.2 : Task spawn trees for two consecutive iterations in SOR

different caches. For this particular problem size, since the data for each place fits into the L2 cache, the cache misses between iterations can be completed removed in SLAW by assigning locality hint to tasks.

Figure 6.3 illustrate the sample code that illustrates the usage of locality hint. At line 1, the programmer gets the runtime constant of the number of places. For each iteration, one top level task is created for each place at line 5. For place p, top level task at place p will then create tasks at line 8. The locality hints of those tasks are inherited from the place that spawns the task, which is place p.

Figure 6.4 shows the performance results of the SLAW locality-aware scheduler using two locality-aware deployments: one with 8-places and 1 worker per place with a total of 8 workers; the other has 8-places and 2 workers per place with a total of 16 workers. The scheduling policy used for task scheduling with each place is the adaptive schedule. The speedup reported for Cilk++ is also based on the Java-serial version in order to compare the execution time.

The 8-place-8-worker locality-aware scheduling is 2.1× faster than the locality-oblivious scheduling using the adaptive scheduling and the speedup for 8-place-16-worker locality-aware scheduling is 2.6×.

```
1  numPlaces = Runtime.getNumPlaces();
2  for (int iter=0; i<NUM_ITERATIONS; iter++) {
3        finish for (int p = 0; p<numPlaces; p++) {
4        place p=Runtime.getPlace(p);
5        async place(p) {
6                  // Task at place p
7                  for (int i=numTasks * p / numPlaces; i<numTasks * (p+1) /
                       numPlaces; i++) {
8                            async {
9                                    // Task
10                                   ....
11                           }
12                 } } }}
```

Figure 6.3 : HJ code snippet with places as locality hint

6.3 Hierarchical Place Trees

Modern computer systems feature multiple homogeneous or heterogeneous computing units with deep memory hierarchies. Exploitation of data locality at multiple levels of memory hierarchy is critical to achieving scalable parallelism. This section briefly describes the recent work-in-progress on the Hierarchical Place Trees (HPT) model [36] which is the hierarchical extension to the previously discussed flat places model.

In the Hierarchical Place Trees (HPT) model, a memory module, such as a DRAM, cache, or device memory, is abstracted as a *place*, and a memory hierarchy is abstracted as a *place tree*. Places are annotated with attributes to indicate their memory type and size, *e.g.,* memory, cache, scratchpad, register file. A processor core is abstracted as a *worker thread*. In our current HPT model, worker threads can only be attached to leaf nodes in the place tree[1]. Figure 6.5 illustrates the locality-based

[1]In the future, we may relax this restriction and allow worker threads to be attached to internal nodes, so as to model "processor-in-memory" hardware architecture.

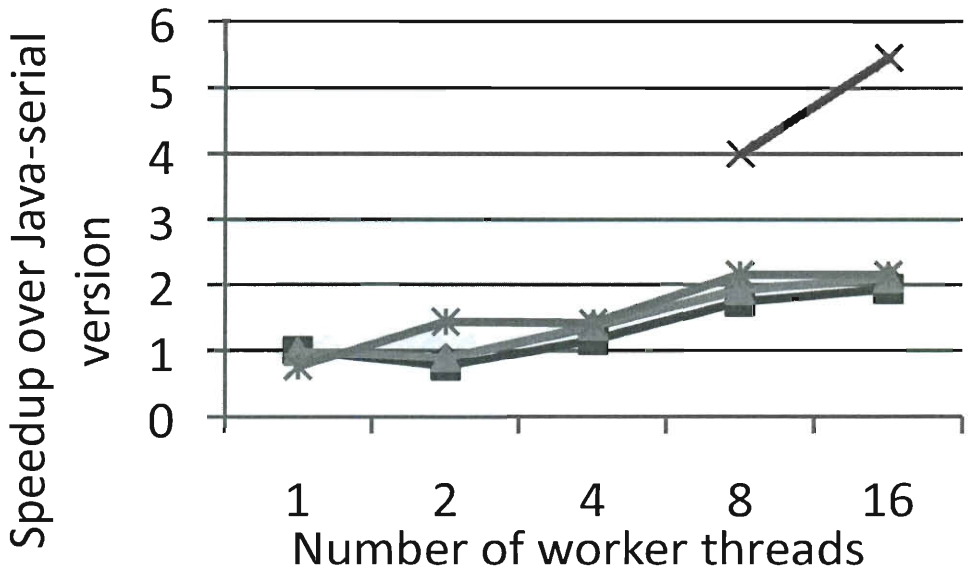


Figure 6.4 : Comparing Locality-aware scheduler with locality-oblivious scheduler on SOR on Intel Xeon SMP. The locality-aware deployment for adp+locality has 8 places with 1 or 2 workers per place. The workers are binded to virtual processors.

scheduling constraints in the HPT model. As in X10, we assume that a task can be directed to place PL_i by using a statement like "*async* (PL_i)". However, unlike X10, the destination place may be an internal node or a leaf node in the hierarchy, as illustrated by the task queues associated with each place in Figure 6.5. If a non-leaf place PL_i is the target for an async statement in the HPT model, then the created task can be executed on any worker that belongs to the subtree rooted at PL_i. Thus, an internal node in the HPT serves as a *subtree wildcard* for the set of workers that can execute a task in its queue. For example, an "*async (PL2)*" task can be executed by worker $w2$ or $w3$. A consequence of this constraint is that a worker can only execute tasks from its ancestor places in the HPT. For example, worker $w0$ in Figure 6.5 can only execute tasks from the queues in places $PL3$, $PL1$, and $PL0$. If a task executing at worker $w0$ is suspended, we assume that it can be resumed at any worker (including $w0$) in the subtree of the task's original target place.

Figure 6.6 illustrates the steps involved in programming and executing an application using the HPT Model. The parallelism and locality in a program is written in

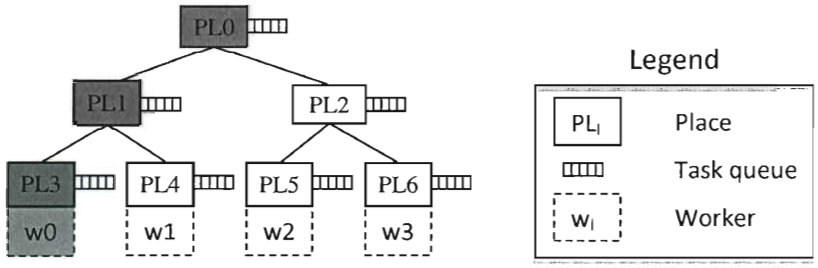


Figure 6.5 : Scheduling constraints in the HPT model

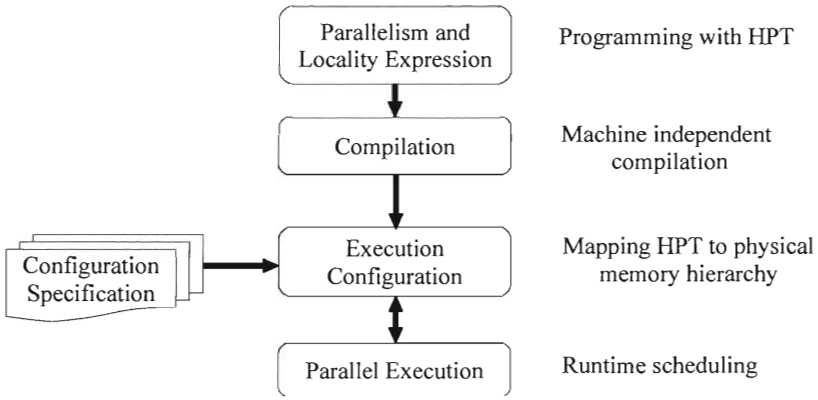


Figure 6.6 : Steps to program and execute an application using the HPT model

a way so as to work with any *configuration specification*. (A configuration consists of an HPT model, and a mapping of the places and workers in the HPT to memories and processor cores in the target machine. This configuration is also called runtime deployment.) Thus, the same program can be executed with different configurations, much as the same OpenMP or MPI program can be executed with different numbers of processors. While it is common to use different configurations as abstractions of different hardware systems, it is also possible to use different configurations as alternate abstractions of the same physical machine. The best configuration choice will depend on both the application and target hardware. Auto-tuning techniques can also be used to help select the best configuration for a specific application and target system.

To illustrate how the HPT model can be used to obtain different abstractions for

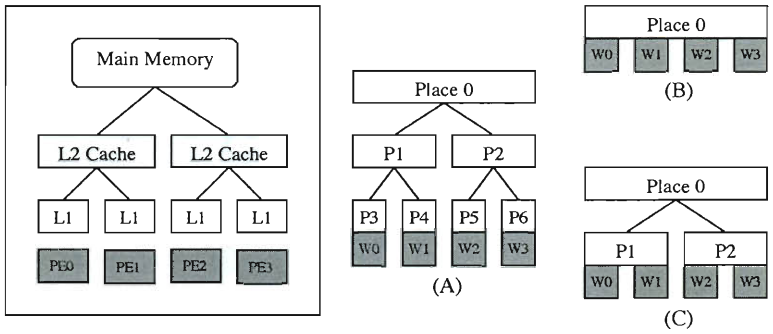


Figure 6.7 : A quad-core CPU machine with a three-level memory hierarchy. Figures a, b, and c represent three different HPT configurations for this machine.

the same physical hardware, consider a quad-core processor machine shown in the left side of Figure 6.7. The hardware consists of four cores (PE0 to PE3) and three levels of memory hierarchy. An HPT model that mirrors this structure can be found on the right in Figure 6.7a. However, if a programmer prefers to view the shared memory as being flat with uniform access, they can instead work with the HPT model shown in Figure 6.7b. Or they can take an intermediate approach by using the HPT model shown in Figure 6.7c.

All data structures that are to be accessed implicitly using global addresses must have a well-defined *distribution* across places. Each scalar object is assumed to have a single *home place*. Any access to any part of the object results in a data transfer from the home place to the worker performing the access. The cost of the access will depend on the distance between the home place and the worker. Note that the programmer, compiler, runtime or hardware may choose to create a cached clone of the object closer to the worker, when legal to do so.

An array can be distributed across multiple places. Unlike a lot of past work on array distributions, the HPT approach to array distribution builds on the idea of *array views* [49]. In this approach, a base one-dimensional array can be allocated across a set of places, and then *viewed* through a multidimensional index space. Multiple views can be created for the same base array, and may range across only a subset of the base array. A key component of an array view is the view's *distribution*, which includes

Name	Description
dist getCartesianView(int rank)	Return a *rank*-dimensional Cartesian view of this place's child places (per-dimension factoring of children is selected by the runtime)
dist getCartesianView(int[ ] dims)	Return a Cartesian view of this place's child places using the per-dimension factors given in the *dims* array
boolean isLeafPlace ()	Return true if this place is a leaf place
Set<place> getChildren()	Return all the child places of this place
placeType getType()	Return the place's storage type (memory, cache, etc)
int getSize ( )	Return the memory size available at this place

Table 6.1 : Subset of place-based API's in the HPT model

the domain and range of the mapping from the view's index space to the base array. We use the `[.]` type notation to denote views and the `[ ]` type notation to denote arrays. Given an array view A, the restriction operation, $A|p$, defines a new array view restricted to elements of A contained within place p's subtree. Note that applying a restriction operator does not result in any data copying or data redistribution. Data transfer only occurs when an array view is dereferenced to access an element of the underlying array.

Table 6.1 lists some of the place-based APIs available to programmers for the HPT model. In Figure 6.8, we show a recursive matrix multiplication program (C=A×B) written in HJ using the HPT interface. There are two portions of code in the example: the code for leaf places executed when the *isLeafPlace( )* predicate evaluates to true, and the code executed for non-leaf places otherwise.

For simplicity, this example only uses implicit data accesses through array views. The views, *A_d*, *B_d* and *C_d*, are used to establish the subregions for recursive calls to *MatrixMult()* via *restriction* operators of the form *A_d*|*p*. As mentioned earlier, creating views does not result in a redistribution of the arrays. Instead, the use of the *ateach* construct in line 17 has the effect of establishing an affinity (akin to tiling) among iterations through the recursive structure of *MatrixMult()*.

```
1   void MatrixMult(double[.] A, double[.] B, double[.] C) {
2       if ( here.isLeafPlace( ) ) {
3           /* compute the sub-block sequentially */
4           for (point [i,j,k] : [A.region.rank(0), B.region.rank(1), A.
                region.rank(1)])
5             C[i,j] += A[i,k] * B[k,j];
6       } else {
7           /* retrieve children places and structure them into a 2-D
                Cartesian topology, pTop */
8           dist pTop = here.getCartesianView( 2 );
9
10          /* generate array view that block-distributes C over the 2-D
                topology, pTop*/
11          final double[.] C\_d = dist.block( C, pTop );
12          /* generate array view that block-distributes A over pTop's 1
                st dimension (rows) */
13          final double[.] A\_d = dist.block( A, pTop, 0 );
14          /* generate array view that block-distributes B over pTop's 2
                nd dimension (columns) */
15          final double[.] B\_d = dist.block( B, pTop, 1 );
16
17          /* recursive call with sub-matrices of A, B, C projected on to
                place p */
18          finish ateach( point p : pTop ) MatrixMult( A\_d|p, B\_d|p, C\
                _d|p );
19      }
20  }
```

Figure 6.8 : Matrix multiplication example

The configuration specification is supplied as an XML file, and describes the target machine architecture as a physical place tree (PPT) as well as a mapping of the HPT to the PPT. Figure 6.9 shows the PPT specification for the quad-core workstation shown in Figure 6.7. In our approach, the mapping is performed when launching

```
1   <ppt:Place id="0" type="memory" xmlns:ppt="http://habanero.rice.edu/pptl" ... >
2       <ppt:Place id="1" type="cache" size="6291456" unitSize="128"> <!-- L2 cache -->
3           <ppt:Place id="3" type="cache" cpuid="0">
4               <ppt:Worker id="0" cpuid="0"/> </ppt:Place>
5           <ppt:Place id="4" type="cache" cpuid="1">
6               <ppt:Worker id="1" cpuid="1"/> </ppt:Place> </ppt:Place>
7       <ppt:Place id="2" type="cache" size="6291456" unitSize="128"> <!-- L2 cache -->
8           <ppt:Place id="5" type="cache" cpuid="2">
9               <ppt:Worker id="2" cpuid="2"/> </ppt:Place>
10          <ppt:Place id="6" type="cache" cpuid="3">
11              <ppt:Worker id="3" cpuid="3"/> </ppt:Place> </ppt:Place>
12  </ppt:Place>
```

Figure 6.9 : Physical place tree specification for a quad-core workstation

the program. This is different from the Sequoia approach in where the mapping is performed by the compiler, thereby requiring a recompilation to generate code for each new hardware configuration.

In Figure 6.9, the *type* attribute is used to specify the type (memory, cache, or accelerator) of the memory module the place represents. The *size* attribute specifies the place's storage size (cache or memory). The *cpuid* attribute is only valid for a worker and is used as a target for mapping HPT worker threads.

Chapter 7

Related Work

The first section of this chapter compare the features of SLAW with those of popular task scheduling systems. In the second section, we discuss recent research efforts in the area of multi-core task scheduling.

7.1 Review of some task scheduling systems

Table 7.1 summarizes the comparison of some popular task scheduling systems.

Cilk/Cilk++

Cilk is a C-based dynamic task parallel language. The Cilk runtime [39] is based on the work-stealing algorithm introduced by Blumofe and Leiserson [15] and is time, space and communication efficient.

Cilk uses the work-first policy for all spawned tasks.The Cilk compiler produces code to support continuation in the function. The stack bound for Cilk is the same as the serial depth-first execution S_1. For a P processor execution, the memory requirement is at most $P \times S_1$.

Cilk does not have support for affinity.

Cilk++ is a recent extension to Cilk and is based on C++ programming language [9]. Cilk++ relaxes the strict parallel calling convection and no longer distinguishes cilk and serial functions. The differences between Cilk++ and Cilk mostly lie in language features while the task scheduling strategy remains unchanged.

System	Type	Cont. Support	Greedy	Heap Bound	Stack Bound	Scheduling Policy	Affinity
Cilk/Cik++	Lang	Yes	Yes	Yes	S_1	Work-first	No
StackThreads	Lib	Yes	Yes	Yes	No	Work-first	No
TBB	Lib	No	No	Yes	S_1	Help-first	Yes
Fork-Join	Lib	No	Yes	-	No	Help-first	No
X10	Lang	No	Yes	No	$O(1)$	Work-sharing	Yes
HJ	Lang	Yes	Yes	Yes	Yes	Adaptive	Yes

Table 7.1 : Comparison of Task Parallel Systems

StackThread/MP

StackThreads/MP is a library that supports fine-grain multi-threading in GCC/G++. StackThreads/MP uses a scheduling scheme that is a combination of work-sharing and work-stealing. In StackThreads/MP, the idle workers send steal requests to busy workers. A steal request can be picked up by the busy worker through the polling routine inserted in the program. Once a steal request is picked up, the victim serves the request by preparing the execution context for the thief. After the context is ready, the thief is notified and begins execution.

As a library, StackThreads/MP has a very unique stack management model to allow programs to compile with standard GCC, which has a standard calling convention, and at the same time, allow task suspension, resume and migration. In StackThreads/MP's stack model, each worker thread has a logical stack as well as a physical stack. One worker's logical stack frames may be spread in the physical stacks of all workers. The model is carefully implemented so that the logical stack can grow and shrink when a function is called and returned using the standard gcc calling convention. StackThreads/MP executes the new task eagerly through a sequential call, which will grow the logical stack and stack of the worker. Task suspension, task resume and task migration and are all implemented by linking the frames of

the worker's local stack. For example, when a task is suspended, the logical stack frames are unwind and saved as continuation. The continuation is essentially a chain of stack frames, is used to resume execution. When logical frames are unwind, they still remain on the physical stack. It is known that StackThreads/MP can have fragmentation in the physical stack and can overflow the stack for large applications.

Intel Thread Building Block

Intel Thread Building Blocks (also known as TBB) is a C++ template library developed by Intel Corporation for writing software programs that take advantage of multicore processors. Unlike StackThread/MP in which continuation is constructed upon task suspension, both TBB and the C compiler do not have continuation support. When a task is suspended and the worker goes stealing, the old stack frames remain on the runtime stack. As a result, the suspended task can only be resumed by the same worker that suspends the task; TBB also restricts the stolen task to those deeper in the spawn tree than the suspended task in order to avoid stack over now in the worst case. These two restrictions will reduce the efficiency the work-stealing load balancing. It has been shown that the depth-restriction can asymptotically serializes execution while unrestricted work-stealing achieves linear speedup [83].

TBB allows the programmer to manually create continuation tasks. However, this approach is essentially trading productivity for performance. In contrast to the Cilk's work-first execution of all spawned tasks, TBB uses the help-first policy upon task creation.

Intel TBB has the `affinity_partitioner` structure to utilize temporal cache-reuse by binding the same iteration to the same worker thread that previously executed the iteration. TBB allows stealing regardless of the affinity and has a mechanism to reduce counter-productive stealing.

Java Fork-Join Framework

Doug Lea's fork join framework is a Java library for fork-join style recursive parallelism. Similar to TBB, Fork Join framework does not have continuation support. When a task is suspended and the worker goes stealing, its stack frames are not cleared. The stolen task's activation frame is pushed on top of the old stack frames. There is no stack bound.

There is no affinity control for tasks in the Fork-Join Framework.

X10

X10 version 1.5 is a Java-based dynamic task parallel language based on PGAS model. In X10, the whole address space is partitioned to places. All data and tasks have affinity to one place. In pure-X10, strict data access rule is enforced: a task can only access local data within its place, otherwise, a `BadPlaceException` would be thrown. Its task scheduling scheme is based on work-sharing. In current X10 implementation, tasks are submitted to a centralized task pool, which is implemented as a ThreadPoolExecutor [10]. There is no continuation support in X10 compiler. When a task is suspended, a new worker thread is created to retrieve tasks from the task pool. This implementation is not scalable.

7.2 Research in Task Parallelism

Some researches in task parallelism focuses on reducing the task scheduling overhead. Hiraishi et al. proposed a backtracking-based work-stealing scheduler called Tascell [45]. In Tascell, program runs normally in sequential mode and backtracks upon a steal request. Tascell trades productivity for performance in programs that have ample parallelism and very few steals. In order to be able to back track, the programmers are given the burden to write roll-back code for each parallel function at the language level. Recent development on Tascell includes an adaptive compilation

strategy that can adaptively switching between high overhead and low overhead version of code [89].

There is similar works in the OpenMP community. Duran et al. compared depth-first and breadth-first task spawning policy in OpenMP task scheduling and found that depth-first performed slightly better than the breadth-first policy [31]. Their breadth-first policy is different from the help-first policy in work-stealing because it uses a global task pool to store all untied tasks, while work-stealing uses local pool per worker. Second, the benchmarks they used to evaluate the performance are mostly task recursive parallel programs where steals are rare.

Another way to reduce the task scheduling overhead is to avoid creating tasks that are too fine-grain. Dural et al. proposed an adaptive cut-off strategy in OpenMP to avoid creating sub tasks that are deep in the task spawn tree [30]. In loop parallelism, chunking techniques are used to reduce the overhead [80]. This kinds of approach has the potential risk of come up with tasks that are too coarse-grain such that the load balancing problem arises again. Some techniques are proposed to find the best granularity [90, 88]. SLAW currently does not change the granularity of the task at runtime. Instead, it tries to schedule the given tasks in an efficient way by policy adaptation. The technique to improve performance by changing the granularity of the tasks are complimentary to SLAW.

Michael et al. proposed a deque implementation for idempotent work-stealing [64], which allows one task to be executed more than once, as long as no task is lost. In idempotent work-stealing, the deque can be implemented more efficiently than the traditional ABP deque [8] because the store-load barrier in the pop and steal operations are removed. This technique is for programmers that are aware of the possible duplicate task executions or for applications whose correctness holds regardless of duplicate task executions.

The locality issue in multithreaded computation has also received a lot of attention in past work [3, 19, 51, 61, 91, 72, 13, 12].

When the number of steals is small, Blumofe's work-stealing algorithm will execute most tasks in the same order as if it were in the sequential execution. It is believed that there is inherent data locality in the sequential execution of a program [14, 67, 3].

Acar et. al presents the theoretical lower and upper bounds on the number of cache misses of Blumofe's work-stealing algorithm on the hardware-controlled shared memory machines [3]. They also presents a locality-guided work-stealing algorithm that improves the data locality using task affinity. Their algorithm assumes each processor has an exclusive cache and does not consider the situation that multiple cores share a L2 cache. The same restriction applies to TBB's `affinity_partitioner` construct. Chen et al.[22] studied and compared the cache behavior between work-stealing and parallel depth-first scheduler on simulators for cores that share the L2 cache. They proposed approaches to control the task granularity and promote constructive cache sharing.

In SLAW, there can be multiple workers under one place, which can be mapped to multiple SMPs. A place can be used to represent a processor or core-pairs that share caches. The workers within a place represents the computing nodes, i.e., the cores. This model can be used to enable constructive cache sharing on a single multicore processor.

With the memory hierarchy increases in depth [34], hierarchical place tree has been proposed to represents the memory hierarchy [36].

In task parallelism, it is both possible for multiple threads working on a large scale of data, or have them working on a small set of data and do lots of computation. The locality-aware framework of SLAW provides a runtime foundation and a tool for programmers or compilers to exploit locality if they wish. However, the approach to exploit the locality depends on the data access pattern of the particular application and is not the focus of this thesis.

A cache-oblivious algorithm is an algorithm designed to exploit the CPU cache without having the size of the cache as an explicit parameter [38]. A cache-oblivious

algorithm typically works in a recursive divide and conquer style. The problem is recursively divided into subproblems, until the subproblems fits into cache. Programs written in cache-oblivious algorithm are insensitive to the underlying cache structure.

There is also some theoretical work regarding the time and space bound of a task scheduling system. Two approaches in past work have been shown to be provably space-efficient. One category consists of work-stealing schedulers with the work-first policy, which were first proven to be space-efficient for fully-strict computations [15]. The same result was later extended for terminally-strict computations [5] in languages like X10 and HJ. Another category of techniques is based on depth-first schedulers [14] such as DFDeques [67], which can use less memory than work-stealing schedulers. Although the parallel depth-first scheduler has a smaller theoretical bound than the work-stealing scheduler, it does not enjoy the practical advantages that the work-stealing algorithm have. Especially, the parallel depth-first scheduler has the problem of high contention among threads due to the global shared structure used to share tasks (similar to work-sharing) and poor locality if the task are too fine grain. All these scheduling techniques focus on the memory space usage without bounding the stack pressure of individual processors, because all models assume that a serial depth-first schedule can run successfully. SLAW's adaptive scheduling algorithm addresses this problem by tracking the stack pressure and generating schedules with bounded stack usage, even in cases when a sequential execution cannot run successfully.

Some research increases the productivity and the expressiveness of the task parallelism by adding new features. Phaser is a new coordination construct that unifies collective and point-to-point synchronizations [79, 78]. Some researches study the work-stealing on a more general task graph that cannot be expressed by the Cilk and HJ's fork-join style parallelism [6, 50]. Hyperobjects, reducers [37] or accumulators [78] are constructs used to collect results from multiple tasks. Supporting new constructs in work-stealing in considered future work.

There are researches that are not based on task parallelism but is related to task

parallelism.

Google's MapReduce [27] is a data parallel programming model design for simple but large scale data processing. The model has also been implemented on both distributed [27, 2] and multicore environment [76]. From the task parallelism perspective, the execution of a MapReduce application consists of two phases. The map phase creates map tasks and the reduce phase creates reduce tasks. There is a strict synchronization barrier between the map and reduce phase.

Galois [53] is a parallel model designed for irregular data parallelism [52]. Its runtime executes task speculatively with conflict controls [62].

Concurrent Collection(CnC) [17] is a high-level data-driven programming model to allow programmers to write code that will run in parallelism while ignoring the low-level threading constructs or scheduling issues. The high-level CnC model is currently converted by the compiler to task parallel model and runs on task parallel runtimes.

Chapter 8

Conclusion

The task parallel programming model and the work-stealing scheduler are increasingly popular, and are considered a promising approach to address the software challenge in the ongoing trend for massive parallelism.

In this dissertation, we describe the implementation of the work-stealing scheduler, SLAW, for Habanero-Java programming language. SLAW supports both work-first and help-first task scheduling policies simultaneously and it comes with compiler support. SLAW features policy adaptation and locality-aware scheduling frame. We theoretically and experimentally show that policy adaptation can deliver performance and resource bound that cannot be achieved by a fixed scheduling policy. We also show the design of SLAW's locality-aware scheduling framework and show an example to use the locality-aware framework to deliver better performance than randomized work-stealing.

The experimental results for the benchmarks studied in this paper show that SLAW's adaptive scheduler achieves 0.98× - 8.9× speedup over the help-first scheduler and 0.97× - 1.56× speedup over the work-first scheduler for 64-thread executions, thereby establishing the robustness of using an adaptive approach instead of a fixed policy. Further, for large irregular recursive parallel computations, the adaptive scheduler runs with bounded stack usage and achieves scalability that cannot be achieved by the use of any single policy. Our experimental results show that locality-aware scheduling can achieve up to 2.59× speedup over locality-oblivious scheduling, for the benchmarks studied in this paper.

The robustness of SLAW on different kinds of parallel applications makes it a good

fit for irregular parallelism as well as a robust foundation to provide general support for higher level programming model such as Futures, Phasers [79] and Reducers [37].

Bibliography

[1] Habanero-Java Web Page. http://habanero.rice.edu/hj.

[2] Hadoop: Open source implementation of mapreduce. http://lucene.apache.org/hadoop/.

[3] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM.

[4] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[5] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM.

[6] Kunal Agrawal, Charles Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[7] John R. Allen and Ken Kennedy. Automatic loop interchange. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 233–246, New York, NY, USA, 1984. ACM.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, New York, NY, USA, 1998. ACM.

[9] Cilk Arts. http://www.cilk.com.

[10] Rajkishore Barik, Vincent Cave, Christopher Donawa, Allan Kielstra, Igor Peshansky, and Vivek Sarkar. Experiences with an smp implementation for x10 based on the java concurrency utilities (extended abstract). In *Proceedings of the 2006 Workshop on Programming Models for Ubiqui- tous Parallelism, co-located with PACT 2006, September 2006, Seattle, Washington*, 2006.

[11] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, 2000.

[12] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[13] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244, New York, NY, USA, 2004. ACM.

[14] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. pages 1–12, 1995.

[15] R. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[16] Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla, and Mateo Valero. A dynamic scheduler for balancing hpc applications. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[17] Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC'09*.

[18] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM.

[19] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in cool. *SIGPLAN Not.*, 28(7):249–259, 1993.

[20] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[21] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM.

[22] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas,

Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[23] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Cray. http://chapel.cray.com/.

[25] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach.* Elsevier Science, 1998.

[26] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7:11–24, April 1988.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[28] Edsger W. Dijkstra. Cooperating sequential processes. pages 65–138, 2002.

[29] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[30] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[31] Alejandro Duran, Julita Corbaln, and Eduard Ayguad. Evaluation of openmp task scheduling strategies. In *International Workshop on OpenMP '08.*

[32] Kemal Ebcioglu, Vivek Sarkar, Tarek El-Ghazawi, and John Urbanic. An experiment in measuring the productivity of three parallel programming languages. In *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2006.

[33] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specification v1.1.1, October 2003.

[34] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[35] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[36] Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. Yonghong yan and jisheng zhao and yi guo and vivek sarkar. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing, LCPC'09*, 1998.

[37] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90, New York, NY, USA, 2009. ACM.

[38] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual*

Symposium on Foundations of Computer Science, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[39] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[40] NANOS group. The Barcelona OpenMP Task Suite (BOTS) Project. http://nanos.ac.upc.edu/projects/bots/wiki.

[41] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[42] Robert H. Halstead. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[43] R. Haralick and G. Elliott. Improving tree search efficiency for constraint-satisfaction problems. *Artificial Intelligence*, 1980.

[44] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289, New York, NY, USA, 2002. ACM.

[45] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 55–64, New York, NY, USA, 2009. ACM.

[46] Steven Hofmeyr, Costin Iancu, and Filip Blagojević. Load balancing on speed. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles*

and practice of parallel programming, pages 147–158, New York, NY, USA, 2010. ACM.

[47] Costin Iancu, Steven Hofmeyr, Filip Blagojevic, and Yili Zheng. Oversubscription on multicore processors. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[48] Intel Corporation. *Intel(R) Threading Building Blocks.*

[49] Mackale Joyner. *Array Optimizations for High Productivity Programming Languages.* PhD thesis, Houston, TX, USA, 2008.

[50] Prabhanjan Kambadur, Anshul Gupta, Amol Ghoting, Haim Avron, and Andrew Lumsdaine. Pfunc: modern task parallelism for modern high performance computing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[51] Vijay Karamcheti and Andrew A. Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17, Washington, DC, USA, 1998. IEEE Computer Society.

[52] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *International Symposium on Performance Analysis of Software and Systems (ISPASS)*, 2009.

[53] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[54] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[55] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[56] James R. Larus and Ravi Rajwar. *Transactional Memory.* Morgan and Claypool, 2006.

[57] D. Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

[58] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 227–242, New York, NY, USA, 2009. ACM.

[59] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. *SIGPLAN Not.*, 44(4):65–74, 2009.

[60] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[61] Evangelos P. Markatos and Thomas J. Leblanc. Locality-based scheduling in shared-memory multiprocessors. Technical report, Parallel Computing: Paradigms and Applications, 1993.

[62] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP '10:*

Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 3–14, New York, NY, USA, 2010. ACM.

[63] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM.

[64] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, New York, NY, USA, 2009. ACM.

[65] Sun Microsystems. http://projectfortress.sun.com/Projects/Community.

[66] Gordon E. Moore. Cramming more components onto integrated circuits. 1965.

[67] G. J. Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of SPAA'99*.

[68] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum Archive*, 17:1–31, August 1998.

[69] OpenMP: A Proposed Industry Standard API for Shared Memory Programming, October 1997. White paper on OpenMP initiative, available at http://www.openmp.org/openmp/mp-documents/paper/paper.ps.

[70] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*, May 2008.

[71] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[72] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread scheduling for cache locality. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 60–71, New York, NY, USA, 1996. ACM.

[73] W. Pugh. Java Memory Model and Thread Specification Revision, 2004. JSR 133, http://www.jcp.org/en/jsr/detail?id=133.

[74] William Pugh. The JAVA memory model is fatally flawed. *Concurrency:Practice and Experience*, 12(1):1–11, 2000.

[75] Raghavan Raman. Compiler support for work-stealing parallel runtime systems, May 2009.

[76] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[77] William N. Scherer, III. *Synchronization and concurrency in user-level software systems.* PhD thesis, Rochester, NY, USA, 2006. Adviser-Scott, Michael L.

[78] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[79] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[80] Jun Shirako, Jisheng M. Zhao, V. Krishna Nandivada, and Vivek N. Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 181–192, New York, NY, USA, 2009. ACM.

[81] Anthony Skjellum, Ewing Lusk, and William Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface.* MIT Press, 1999.

[82] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 91–100, New York, NY, USA, 2009. ACM.

[83] Jim Sukha. Brief announcement: a lower bound for depth-restricted work stealing. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 124–126, New York, NY, USA, 2009. ACM.

[84] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[85] Xavier Teruel, Priya Unnikrishnan, Xavier Martorell, Eduard Ayguadé, Raul Silvera, Guansong Zhang, and Ettore Tiotto. Openmp tasks in ibm xl compilers. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 207–221, New York, NY, USA, 2008. ACM.

[86] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, 1967.

[87] Philippas Tsigas and Yi Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 29(1):320–321, 2001.

[88] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, New York, NY, USA, 2010. ACM.

[89] Lei Wang, Huimin Cui, Yuelu Duan, Fang Lu, Xiaobing Feng, and Pen-Chung Yew. An adaptive task creation strategy for work-stealing scheduling. In *CGO '10: Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2010. IEEE Computer Society.

[90] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 75–84, New York, NY, USA, 2009. ACM.

[91] Boris Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 127–138, New York, NY, USA, 1998. ACM.

[92] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, New York, NY, USA, 1991. ACM.

[93] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and

Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.